Type-Theoretic Signatures for Algebraic Theories and Inductive Types

András Kovács

2021 September

Contents

CHAPTER 1

Introduction

1.1 Specification and Semantics for Inductive Types

1.2 Overview of the Thesis and Contributions

1.3 Notation and Conventions

1.3.1 Metatheory

1.3.2 Universes

CHAPTER 2

Simple Inductive Signatures

In this chapter, we take a look at a very simple notion of inductive signature. The motivation for doing so is to present the basic ideas of this thesis in the easiest possible setting, with explicit definitions. The later chapters are greatly generalized and expanded compared to the current one, and are not feasible (and probably not that useful) to present in full formal detail. We also include a complete Agda formalization of the contents of this chapter, in less than 200 lines.

potentially in intro

The mantra throughout this dissertation is the following: inductive types are specified by typing contexts in certain *theories of signatures*. For each class of inductive types, there is a corresponding theory of signatures, which is viewed as a proper type theory and comes equipped with an algebraic model theory. *Semantics* of signatures is given by interpreting them in certain models of the theory of signatures. Semantics should at least provide a notion of induction principle for each signature; in this chapter we provide a bit more than that, and substantially more in Chapters 4 and 5.

2.1 Theory of Signatures

Generally, more expressive theories of signatures can describe a larger classes of inductive types. As we are aiming at minimalism right now, the current theory of signatures is as follows:

Definition 1. The **theory of signatures**, or ToS for short, is a simple type theory equipped with the following features:

- An empty base type ι.

- A *first-order function type* $\iota \to -$; this is a function whose domain is fixed to be ι. Moreover, first-order functions only have neutral terms: there is application, but no λ-abstraction.

We can specify the full syntax using the following Agda-like inductive definitions.

$$
\begin{array}{ll}
\mathsf{Ty} \quad : \mathsf{Set} & \mathsf{Var} : \mathsf{Con} \to \mathsf{Ty} \to \mathsf{Set} \\
\iota \quad : \mathsf{Ty} & \mathsf{vz} \quad : \mathsf{Var}\,(\Gamma \triangleright A)\,A \\
\iota \to - : \mathsf{Ty} \to \mathsf{Ty} & \mathsf{vs} \quad : \mathsf{Var}\,\Gamma\,A \to \mathsf{Var}\,(\Gamma \triangleright B)\,A
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{Con} \quad : \mathsf{Set} & \mathsf{Tm} : \mathsf{Con} \to \mathsf{Ty} \to \mathsf{Set} \\
\bullet \quad : \mathsf{Con} & \mathsf{var} \; : \mathsf{Var}\,\Gamma\,A \to \mathsf{Tm}\,\Gamma\,A \\
- \triangleright - : \mathsf{Con} \to \mathsf{Ty} \to \mathsf{Con} & \mathsf{app} : \mathsf{Tm}\,\Gamma\,(\iota \to A) \to \mathsf{Tm}\,\Gamma\,\iota \to \mathsf{Tm}\,\Gamma\,A
\end{array}
$$

Here, Con contexts are lists of types, and Var specifies well-typed De Bruijn indices, where vz represents the zero index, and vs takes the successor of an index.

Notation 1. We use capital Greek letters starting from Γ to refer to contexts, A, B, C to refer to types, and t, u, v to refer to terms. In examples, we may use a nameful notation instead of De Bruijn indices. For example, we may write $x : \mathsf{Tm}\,(\bullet \triangleright (x : \iota) \triangleright (y : \iota))\,\iota$ instead of $\mathsf{var}\,(\mathsf{vs}\,\mathsf{vz}) : \mathsf{Tm}\,(\bullet \triangleright \iota \triangleright \iota)\,\iota$. Additionally, we may write $t\,u$ instead of $\mathsf{app}\,t\,u$ for t and u terms.

Definition 2. Parallel substitutions map variables to terms.

$$
\begin{array}{l}
\mathsf{Sub} : \mathsf{Con} \to \mathsf{Con} \to \mathsf{Set} \\
\mathsf{Sub}\,\Gamma\,\Delta \equiv \{A\} \to \mathsf{Var}\,\Delta\,A \to \mathsf{Tm}\,\Gamma\,A
\end{array}
$$

We use σ and δ to refer to substitutions. We also recursively define the action of substitution on terms:

$$
\begin{array}{l}
-[-] : \mathsf{Tm}\,\Delta\,A \to \mathsf{Sub}\,\Gamma\,\Delta \to \mathsf{Tm}\,\Gamma\,A \\
(\mathsf{var}\,x) \;\; [\sigma] :\equiv \sigma\,x \\
(\mathsf{app}\,t\,u)[\sigma] :\equiv \mathsf{app}\,(t[\sigma])\,(u[\sigma])
\end{array}
$$

The identity substitution id is defined simply as var. It is easy to see that $t[\mathsf{id}] = t$ for all t. Substitution composition is as follows.

$$-\circ- : \mathsf{Sub}\,\Delta\,\Xi \to \mathsf{Sub}\,\Gamma\,\Delta \to \mathsf{Sub}\,\Gamma\,\Xi$$
$$(\sigma \circ \delta)\,x :\equiv (\sigma\,x)[\delta]$$

Example 1. We may write signatures for natural numbers and binary trees respectively as follows.

$$\mathsf{NatSig}\ :\equiv \bullet \triangleright (zero : \iota) \triangleright (suc : \iota \to \iota)$$
$$\mathsf{TreeSig} :\equiv \bullet \triangleright (leaf : \iota) \triangleright (node : \iota \to \iota \to \iota)$$

In short, the current ToS allows inductive types which are

- *Single-sorted*: this means that we have a single type constructior, corresponding to ι.

- *Closed*: signatures cannot refer to any externally existing type. For example, we cannot write a signature for "lists of natural number" in a direct fashion, since there is no way to refer to the type of natural numbers.

- *Finitary*: inductive types corresponding to signatures are always finitely branching trees. Being closed implies being finitary, since an infinitely branching node would require some external type to index subtrees with. For example, $node : (\mathbb{N} \to \iota) \to \iota$ would specify an infinite branching (if such type was allowed in ToS).

Remark. We omit λ-expressions from ToS for the sake of simplicity: this causes terms to be always in normal form (neutral, to be precise), and thus we can skip talking about conversion rules. Later, starting from Chapter 4 we include proper $\beta\eta$-rules in signature theories.

2.2   Semantics

For each signature, we need to know what it means for a type theory to support the corresponding inductive type. For this, we need at least a notion of *algebras*, which can be viewed as a bundle of all type and value constructors, and what it means for an algebra to support an *induction principle*. Additionally, we may

want to know what it means to support a *recursion principle*, which can be viewed as a non-dependent variant of induction. In the following, we define these notions by induction on ToS syntax.

Remark. We use "algebra" and "model" synonymously throughout this thesis.

2.2.1 Algebras

First, we calculate types of algebras. This is simply a standard interpretation into the Set universe. We define the following operations by induction; the $-^A$ name is overloaded for Con, Ty and Tm.

$$-^A : \mathsf{Ty} \to \mathsf{Set} \to \mathsf{Set}$$
$$\iota^A \qquad X :\equiv X$$
$$(\iota \to A)^A\, X :\equiv X \to A^A\, X$$

$$-^A : \mathsf{Con} \to \mathsf{Set} \to \mathsf{Set}$$
$$\Gamma^A\, X :\equiv \{A : \mathsf{Ty}\} \to \mathsf{Var}\, \Gamma\, A \to A^A\, X$$

$$-^A : \mathsf{Tm}\, \Gamma\, A \to \{X : \mathsf{Set}\} \to \Gamma^A\, X \to A^A\, X$$
$$(\mathsf{var}\, x)^A \quad \gamma :\equiv \gamma\, x$$
$$(\mathsf{app}\, t\, u)^A\, \gamma :\equiv t^A\, \gamma\, (u^A\, \gamma)$$

$$-^A : \mathsf{Sub}\, \Gamma\, \Delta \to \{X : \mathsf{Set}\} \to \Gamma^A\, X \to \Delta^A\, X$$
$$\sigma^A\, \gamma\, x :\equiv (\sigma\, x)^A\, \gamma$$

Here, types and contexts depend on some $X : \mathsf{Set}$, which serves as the interpretation of ι. We define Γ^A as a product: for each variable in the context, we get a semantic type. This trick, along with the definition of Sub, makes formalization a bit more compact. Terms and substitutions are interpreted as natural maps. Substitutions are interpreted by pointwise interpreting the contained terms.

Notation 2. We may write values of Γ^A using notation for Σ-types. For example, we may write $(zero : X) \times (suc : X \to X)$ for the result of computing $\mathsf{NatSig}^A\, X$.

Definition 3. We define **algebras** as follows.

$$\mathsf{Alg} : \mathsf{Con} \to \mathsf{Set}_1$$
$$\mathsf{Alg}\,\Gamma :\equiv (X : \mathsf{Set}) \times \Gamma^A\,X$$

Example 2. $\mathsf{Alg}\,\mathsf{NatSig}$ is computed to $(X : \mathsf{Set}) \times (zero : X) \times (suc : X \to X)$.

2.2.2   Morphisms

Now, we compute notions of morphisms of algebras. In this case, morphisms are functions between underlying sets which preserve all specified structure. The interpretation for calculating morphisms is a *logical relation interpretation* [HRR14] over the $-^A$ interpretation. The key part is the interpretation of types:

$$-^M : (A : \mathsf{Ty})\{X_0\,X_1 : \mathsf{Set}\}(X^M : X_0 \to X_1) \to A^A\,X_0 \to A^A\,X_1 \to \mathsf{Set}$$
$$\iota^M \qquad\quad X^M\,\alpha_0\,\alpha_1 :\equiv X^M\,\alpha_0 = \alpha_1$$
$$(\iota \to A)^M\,X^M\,\alpha_0\,\alpha_1 :\equiv (x : X_0) \to A^M\,X^M\,(\alpha_0\,x)\,(\alpha_1\,(X^M\,x))$$

We again assume an interpretation for the base type ι, as X_0, X_1 and $X^M : X_0 \to X_1$. X^M is function between underlying sets of algebras, and A^M computes what it means that X^M preserves an operation with type A. At the base type, preservation is simply equality. At the first-order function type, preservation is a quantified statement over X_0. We define morphisms for Con pointwise:

$$-^M : (\Gamma : \mathsf{Con})\{X_0\,X_1 : \mathsf{Set}\} \to (X_0 \to X_1) \to \Gamma^A\,X_0 \to \Gamma^A\,X_1 \to \mathsf{Set}$$
$$\Gamma^M\,X^M\,\gamma_0\,\gamma_1 :\equiv \{A : \mathsf{Ty}\}(x : \mathsf{Var}\,\Gamma\,A) \to A^M\,X^M\,(\gamma_0\,x)\,(\gamma_1\,x)$$

For terms and substitutions, we get preservation statements, which are sometimes called *fundamental lemmas* in discussions of logical relations [HRR14].

$$-^M : (t : \mathsf{Tm}\,\Gamma\,A) \to \Gamma^M\,X^M\,\gamma_0\,\gamma_1 \to A^M\,X^M\,(t^A\,\gamma_0)\,(t^A\,\gamma_1)$$
$$(\mathsf{var}\,x)^M \quad \gamma^M :\equiv \gamma^M\,x$$
$$(\mathsf{app}\,t\,u)^M \gamma^M :\equiv t^M\,\gamma^M\,(u^A\,\gamma_0)$$

$$-^M : (\sigma : \mathsf{Sub}\,\Gamma\,\Delta) \to \Gamma^M\,X^M\,\gamma_0\,\gamma_1 \to \Delta^M\,X^M\,(\sigma^A\,\gamma_0)\,(\sigma^A\,\gamma_1)$$
$$\sigma^M\,\gamma^M\,x :\equiv (\sigma\,x)^M\,\gamma^M$$

The definition of $(\mathsf{app}\,t\,u)^M$ is well-typed by the induction hypothesis $u^M\,\gamma^M : X^M\,(u^A\,\gamma_0) = u^A\,\gamma_1$.

Definition 4. To get notions of **algebra morphisms**, we again pack up Γ^M with the interpretation of ι.

$$\mathsf{Mor} : \{\Gamma : \mathsf{Con}\} \to \mathsf{Alg}\,\Gamma \to \mathsf{Alg}\,\Gamma \to \mathsf{Set}$$

$$\mathsf{Mor}\,\{\Gamma\}\,(X_0,\,\gamma_0)\,(X_1,\,\gamma_1) :\equiv (X^M : X_0 \to X_1) \times \Gamma^M\,X^M\,\gamma_0\,\gamma_1$$

Example 3. We have the following computation:

$$\mathsf{Mor}\,\{\mathsf{NatSig}\}\,(X_0,\,zero_0,\,suc_0)\,(X_0,\,zero_1,\,suc_1) :\equiv$$
$$(X^M : X_0 \to X_1)$$
$$\times\,(X^M\,zero_0 = zero_1)$$
$$\times\,((x : X_0) \to X^M\,(suc_0\,x) = suc_1\,(X^M\,x))$$

Definition 5. We state **initiality** as a predicate on algebras:

$$\mathsf{Initial} : \{\Gamma : \mathsf{Con}\} \to \mathsf{Alg}\,\Gamma \to \mathsf{Set}$$

$$\mathsf{Initial}\,\{\Gamma\}\,\gamma :\equiv (\gamma' : \mathsf{Alg}\,\Gamma) \to \mathsf{isContr}\,(\mathsf{Mor}\,\gamma\,\gamma')$$

Here $\mathsf{isContr}$ refers to unique existence [Uni13, Section 3.11]. If we drop $\mathsf{isContr}$ from the definition, we get the notion of weak initiality, which corresponds to the recursion principle for Γ. Although we call this predicate $\mathsf{Initial}$, in this chapter we do not yet show that algebras form a category. We provide the extended semantics in Chapter 4. The computed algebras and morphism there remain the same as in the current chapter.

Morphisms vs. logical relations.   The $-^M$ interpretation can be viewed as a special case of logical relations over the $-^A$ model: every morphism is a *functional* logical relation, where the chosen relation between the underlying sets happens to be a function. Consider now a more general relational interpretation for types:

$$-^R : (A : \mathsf{Ty})\{X_0\,X_1 : \mathsf{Set}\}(X^R : X_0 \to X_1 \to \mathsf{Set}) \to A^A\,X_0 \to A^A\,X_1 \to \mathsf{Set}$$

$$\iota^R \qquad X^R\,\alpha_0\,\alpha_1 :\equiv X^R\,\alpha_0\,\alpha_1$$

$$(\iota \to A)^R\,X^R\,\alpha_0\,\alpha_1 :\equiv (x_0 : X_0)(x_1 : X_1) \to X^R\,x_0\,x_1 \to A^R\,X^R\,(\alpha_0\,x_0)\,(\alpha_1\,x_1)$$

Here, functions are related if they map related inputs to related outputs. If we know that $X^M\,\alpha_0\,\alpha_1 \equiv (f\,\alpha_0 = \alpha_1)$ for some f function, we get

$$(x_0 : X_0)(x_1 : X_1) \to f\,x_0 = x_1 \to A^R\,X^R\,(\alpha_0\,x_0)\,(\alpha_1\,x_1)$$

Now, we can simply substitute along the input equality proof in the above type, to get the previous definition for $(\iota \to A)^M$:

$$(x_0 : X_0) \to A^R \, X^R \, (\alpha_0 \, x_0) \, (\alpha_1 \, (f \, x_0))$$

This substitution along the equation is called "singleton contraction" in the jargon of homotopy type theory [Uni13]. The ability to perform contraction here is at the heart of the *strict positivity restriction* for inductive signatures. Strict positivity in our setting corresponds to only having first-order function types in signatures. If we allowed function domains to be arbitrary types, in the definition of $(A \to B)^M$ we would only have a black-box $A^M \, X^M : A^A \, X_0 \to A^A \, X_1 \to \mathsf{Set}$ relation, which is not known to be given as an equality.

In Chapter 4 we expand on this. As a preliminary summary: although higher-order functions have relational interpretation, such relations do not generally compose. What we eventually aim to have is a *category* of algebras and algebra morphisms, where morphisms do compose. We need a *directed* model of the theory of signatures, where every signature becomes a category of algebras. The way to achieve this, is to prohibit higher-order functions, thereby avoiding the polarity issues that prevent a directed interpretation for general function types.

2.2.3   Displayed Algebras

At this point we do not yet have specification for induction principles. We use the term *displayed algebra* to refer to "dependent" algebras, where every displayed algebra component lies over corresponding components in the base algebra. For the purpose of specifying induction, displayed algebras can be viewed as bundles of induction motives and methods.

Displayed algebras over some $\gamma : \mathsf{Alg}\,\Gamma$ are equivalent to slices over γ in the category of Γ-algebras; we show this in Chapter 4. A slice $f : \mathsf{Sub}\,\Gamma\,\gamma'\,\gamma$ maps elements of γ's underlying set to elements in the base algebra. Why do we need displayed algebras, then? The main reason is that if we are to eventually implement inductive types in a dependently typed language, we need to compute induction principles exactly, not merely up to isomorphisms.

For more illustration of using some displayed algebras in a type-theoretic setting, see [AL19]. We adapt the term "displayed algebra" from ibid. as a generalization of displayed categories (and functors, natural transformations) to other algebraic structures.

The displayed algebra interpretation is a *logical predicate* interpretation, defined as follows.

$$-^D : (A : \mathsf{Ty})\{X\} \to (X \to \mathsf{Set}) \to A^A\, X \to \mathsf{Set}$$
$$\iota^D \qquad X^D\, \alpha :\equiv X^D\, \alpha$$
$$(\iota \to A)^D\, X^D\, \alpha :\equiv (x : X)(x^D : X^D\, x) \to A^D\, X^D\, (\alpha\, x)$$

$$-^D : (\Gamma : \mathsf{Con})\{X\} \to (X \to \mathsf{Set}) \to \Gamma^A\, X \to \mathsf{Set}$$
$$\Gamma^D\, X^D\, \gamma :\equiv \{A : \mathsf{Ty}\}(x : \mathsf{Var}\,\Gamma\, A) \to A^D\, X^D\, (\gamma\, x)$$

$$-^D : (t : \mathsf{Tm}\,\Gamma\, A) \to \Gamma^D\, X^D\, \gamma \to A^D\, X^D\, (t^A\, \gamma)$$
$$(\mathsf{var}\, x)^D \quad \gamma^D :\equiv \gamma^D\, x$$
$$(\mathsf{app}\, t\, u)^D\, \gamma^D :\equiv t^D\, \gamma^D\, (u^A\, \gamma)\, (u^D\, \gamma^D)$$

$$-^D : (\sigma : \mathsf{Sub}\,\Gamma\,\Delta) \to \Gamma^D\, X^D\, \gamma \to \Delta^D\, X^D\, (\sigma^A\, \gamma)$$
$$\sigma^D\, \gamma^D\, x :\equiv (\sigma\, x)^D\, \gamma^D$$

Analogously to before, everything depends on a predicate interpretation $X^D : X \to \mathsf{Set}$ for ι. For types, a predicate holds for a function if the function preserves predicates. The interpretation of terms is again a fundamental lemma, and we again have pointwise definitions for contexts and substitutions.

Definition 6 (displayed algebras).

$$\mathsf{DispAlg} : \{\Gamma : \mathsf{Con}\} \to \mathsf{Alg}\,\Gamma \to \mathsf{Set}_1$$
$$\mathsf{DispAlg}\,\{\Gamma\}\,(X, \gamma) :\equiv (X^D : X \to \mathsf{Set}) \times \Gamma^D\, X^D\, \gamma$$

Example 4. We have the following computation.

$$\mathsf{DispAlg}\,\{\mathsf{NatSig}\}\,(X,\, zero,\, suc) \equiv$$
$$(X^D \quad : X \to \mathsf{Set})$$
$$\times (zero^D : X^D\, zero)$$
$$\times (suc^D \;\; : (n : X) \to X^D\, n \to X^D\, (suc\, n))$$

2.2.4   Sections

Sections of displayed algebras are "dependent" analogues of algebra morphisms, where the codomain is displayed over the domain.

$$
\begin{aligned}
&-^S : (A : \mathsf{Ty})\{X\ X^D\}(X^S : (x : X) \to X^D\ x) \to (\alpha : A^A\ X) \to A^D\ X^D\ \alpha \to \mathsf{Set} \\
&\iota^S \qquad\quad X^S\ \alpha\ \alpha^D :\equiv X^S\ \alpha = \alpha^D \\
&(\iota \to A)^S\ X^S\ \alpha\ \alpha^D :\equiv (x : X) \to A^S\ X^S\ (\alpha\ x)\ (\alpha^D\ (X^S\ x))
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{Con}^S : (\Gamma : \mathsf{Con})\{X\ X^D\}(X^S : (x : X) \to X^D\ x) \to (\gamma : \Gamma^A\ X) \to \Gamma^D\ X^D\ \gamma \to \mathsf{Set} \\
&\Gamma^S\ X^S\ \gamma_0\ \gamma_1 :\equiv \{A : \mathsf{Ty}\}(x : \mathsf{Var}\ \Gamma\ A) \to A^S\ X^S\ (\gamma_0\ x)\ (\gamma_1\ x)
\end{aligned}
$$

$$
\begin{aligned}
&-^S : (t : \mathsf{Tm}\ \Gamma\ A) \to \Gamma^S\ X^S\ \gamma\ \gamma^D \to A^S\ X^S\ (t^A\ \gamma)\ (t^D\ \gamma^D) \\
&(\mathsf{var}\ x)^S \quad \gamma^S :\equiv \gamma^S\ x \\
&(\mathsf{app}\ t\ u)^S \gamma^S :\equiv t^S\ \gamma^S\ (u^A\ \gamma)
\end{aligned}
$$

$$
\begin{aligned}
&-^S : (\sigma : \mathsf{Sub}\ \Gamma\ \Delta) \to \Gamma^S\ X^S\ \gamma\ \gamma^D \to \Delta^S\ X^S\ (\sigma^A\ \gamma)\ (\sigma^A\ \gamma^D) \\
&\sigma^S\ \gamma^S\ x = (\sigma\ x)^S\ \gamma^S
\end{aligned}
$$

Definition 7 (Displayed algebra sections ("sections" in short)**).**

$$
\begin{aligned}
&\mathsf{Section} : \{\Gamma : \mathsf{Con}\} \to (\gamma : \mathsf{Alg}\ \Gamma) \to \mathsf{DispAlg}\ \gamma \to \mathsf{Set} \\
&\mathsf{Section}\ (X,\ \gamma)\ (X^D\ \gamma^D) :\equiv (X^S : (x : X) \to X^D\ x) \times \Gamma^S\ X^S\ \gamma\ \gamma^D
\end{aligned}
$$

Example 5. We have the following computation.

$$
\begin{aligned}
\mathsf{Section}\ \{\mathsf{NatSig}\}\ (X,\ zero,\ suc)\ (X^D,\ zero^D,\ suc^D) \equiv& \\
(X^S \quad : (x : X) \to X^D\ x)& \\
\times\ (zero^S : X^S\ zero = zero^D)& \\
\times\ (suc^S \ : (n : X) \to X^S\ (suc\ n) = suc^D\ n\ (X^S\ n))&
\end{aligned}
$$

Definition 8 (Induction). We define a predicate which holds if an algebra supports induction.

$$
\begin{aligned}
&\mathsf{Inductive} : \{\Gamma : \mathsf{Con}\} \to \mathsf{Alg}\ \Gamma \to \mathsf{Set}_1 \\
&\mathsf{Inductive}\ \{\Gamma\}\ \gamma :\equiv (\gamma^D : \mathsf{DispAlg}\ \gamma) \to \mathsf{Section}\ \gamma\ \gamma^D
\end{aligned}
$$

We can observe that $\mathsf{Inductive}\,\{\mathsf{NatSig}\}\,(X, \mathsf{zero}, \mathsf{suc})$ computes exactly to the usual induction principle for natural numbers. The input $\mathsf{DispAlg}$ is a bundle of the induction motive and the methods, and the output $\mathsf{Section}$ contains the X^S eliminator function together with its β computation rules.

2.3 Term Algebras

In this section we show that if a type theory supports the inductive types comprising the theory of signatures, it also supports every inductive type which is described by the signatures.

Note that we specified Tm and Sub, but did not need either of them when specifying signatures, or when computing induction principles. That signatures do not depend on terms, is a property specific to simple signatures; this will not be the case in Chapter 4 when we move to more general signatures. However, terms and substitutions are already useful here in the construction of term algebras.

The idea is that terms in contexts comprise initial algebras. For example, $\mathsf{Tm}\,\mathsf{NatSig}\,\iota$ is the set of natural numbers (up to isomorphism). Informally, this is because the only way to construct terms is by applying the suc variable (given by $\mathsf{var}\,\mathsf{vz}$) finitely many times to the zero variable (given by $\mathsf{var}\,(\mathsf{vs}\,\mathsf{vz})$).

Definition 9 (Term algebras). Fix an $\Omega : \mathsf{Con}$. We abbreviate $\mathsf{Tm}\,\Omega\,\iota$ as T; this will serve as the carrier set of the term algebra. We additionally define the following.

$$-^T : (A : \mathsf{Ty}) \to \mathsf{Tm}\,\Omega\,A \to A^A\,\mathsf{T}$$
$$\iota^T \qquad t :\equiv t$$
$$(\iota \to A)^T\,t :\equiv \lambda\,u.\,A^T\,(\mathsf{app}\,t\,u)$$

$$-^T : (\Gamma : \mathsf{Con}) \to \mathsf{Sub}\,\Omega\,\Gamma \to \Gamma^A\,\mathsf{T}$$
$$\Gamma^T\,\nu\,\{A\}\,x :\equiv A^T\,(\nu\,x)$$

$$-^T : (t : \mathsf{Tm}\,\Gamma\,A)(\nu : \mathsf{Sub}\,\Omega\,\Gamma) \to A^T\,(t[\nu]) = t^A\,(\Gamma^T\,\nu)$$
$$(\mathsf{var}\,x)^T \quad \nu \text{ holds by } \mathsf{refl}$$
$$(\mathsf{app}\,t\,u)^T\,\nu \text{ holds by } t^T\,\nu \text{ and } u^T\,\nu$$

$$-^T : (\sigma : \mathsf{Sub}\,\Gamma\,\Delta)(\nu : \mathsf{Sub}\,\Omega\,\Gamma)\{A\}(x : \mathsf{Var}\,\Delta\,A)$$
$$\to \Delta^T\,(\sigma \circ \nu)\,x = \sigma^A\,(\Gamma^T\,\nu)\,x$$
$$\sigma^T\,\nu\,x :\equiv (\sigma\,x)^T\,\nu$$

Now we can define the term algebra for Ω itself:

$$\mathsf{TmAlg}_\Omega : \mathsf{Alg}\,\Omega$$
$$\mathsf{TmAlg}_\Omega :\equiv \Omega^T\,\Omega\,\mathsf{id}$$

In the interpretation for contexts, it is important that Ω is fixed, and we do induction on all Γ contexts such that there is a $\mathsf{Sub}\,\Omega\,\Gamma$. It would not work to try to compute term algebras by direct induction on contexts, because we need to refer to the same T set in the interpretation of every type in a signature.

The interpretation of types embeds terms as A-algebras. At the base type ι, this embedding is simply the identity function, since $\iota^A\,\mathsf{T} \equiv \mathsf{T} \equiv \mathsf{Tm}\,\Omega\,\iota$. At function types we recursively proceed under a semantic λ. The interpretation of contexts is pointwise.

The interpretations of terms and substitutions are coherence properties, which relate the term algebra construction to term evaluation in the $-^A$ model. For terms, if we pick $\nu \equiv \mathsf{id}$, we get $A^T\,t = t^A\,\mathsf{TmAlg}_\Omega$. The left side embeds t in the term model via $-^T$, while the right hand side evaluates t in the term model.

A way to view the term algebra construction, is that we are working in a *slice model* over the fixed Ω, and every $\nu : \mathsf{Sub}\,\Omega\,\Gamma$ can be viewed as an internal Γ-algebra in this model. The term algebra construction demonstrates that every such internal algebra yields an external element of Γ^A. We will see in Section 45 that we can construct term algebras from *any* model of a ToS, not just the ToS syntax; but while term algebras constructed from ToS syntax are themselves initial algebras, in other cases they may not be initial.

2.3.1   Weak Initiality

We show that TmAlg_Ω supports a recursion principle, i.e. it is weakly initial.

Definition 10 (Recursor construction). We assume $(X, \omega) : \mathsf{Alg}\,\Omega$; recall that $X : \mathsf{Set}$ and $\omega : \Omega^A\,X$. We define $\mathsf{R} : \mathsf{T} \to X$ as $\mathsf{R}\,t \equiv t^A\,\omega$. We additionally

define the following.

$$-^R : (A : \mathsf{Ty})(t : \mathsf{Tm}\,\Omega\,A) \to A^M\,\mathsf{R}\,(A^T\,t)\,(t^A\,\omega)$$
$$\iota^R \qquad t :\equiv (\mathsf{refl} : t^A\,\omega = t^A\,\omega)$$
$$(\iota \to A)^R\,t :\equiv \lambda\,u.\,A^R\,(\mathsf{app}\,t\,u)$$

$$-^R : (\Gamma : \mathsf{Con})(\nu : \mathsf{Sub}\,\Omega\,\Gamma) \to \Gamma^M\,\mathsf{R}\,(\Gamma^T\,\nu)\,(\nu^A\,\omega)$$
$$\Gamma^R\,\nu\,x :\equiv A^R\,(\nu\,x)$$

We define the recursor for Ω as

$$\mathsf{Rec}_\Omega : (alg : \mathsf{Alg}\,\Omega) \to \mathsf{Mor}\,\mathsf{TmAlg}_\Omega\,alg$$
$$\mathsf{Rec}_\Omega\,(X,\,\omega) :\equiv (\mathsf{R},\,\Omega^R\,\Omega\,\mathsf{id})$$

In short, the way we get recursion is by evaluating terms in arbitrary $(X,\,\omega)$ algebras using $-^A$. The $-^R$ operation for types and contexts confirms that R preserves structure appropriately, so that R indeed yields algebra morphisms.

We skip interpreting terms and substitutions by $-^R$. It is necessary to do so with more general signatures, but not in the current chapter.

2.3.2 Induction

We take the idea of the previous section a bit further. We have seen that recursion for term algebras is given by evaluation in the "standard" model $-^A$. Now, we show that induction for term algebras corresponds to evaluation in the logical predicate model $-^D$.

Definition 11 (Eliminator construction). We assume $(X^D,\,\omega^D) : \mathsf{DispAlg}\,\mathsf{TmAlg}_\Omega$. Recall that $X^D : \mathsf{T} \to \mathsf{Set}$ and $\omega^D : \Omega^D\,X^D\,(\Omega^T\,\Omega\,\mathsf{id})$. Like before, we first interpret the underlying set:

$$\mathsf{E} : (t : \mathsf{T}) \to X^D\,t$$
$$\mathsf{E}\,t :\equiv t^D\,\omega^D$$

However, this definition is not immediately well-typed, since $t^D\,\omega^D$ has type $X^D\,(t^A\,(\Omega^T\,\Omega\,\mathsf{id}))$, so we have to show that $t^A\,(\Omega^T\,\Omega\,\mathsf{id}) = t$. This equation says that nothing happens if we evaluate a term with type ι in the term model. We

get it from the $-^T$ interpretation of terms: $t^T\,\mathsf{id} : t[\mathsf{id}] = t^A\,(\Omega^T\,\Omega\,\mathsf{id})$, and we also know that $t[\mathsf{id}] = t$. We interpret types and contexts as well:

$$-^E : (A : \mathsf{Ty})(t : \mathsf{Tm}\,\Omega\,A) \to A^S\,\mathsf{E}\,(t^A\,(\Omega^T\,\Omega\,\mathsf{id}))\,(t^D\,\omega^D)$$

$$\iota^E \qquad t : (t^A\,(\Omega^T\,\Omega\,\mathsf{id}))^D\,\omega^D = t^D\,\omega^D$$

$$(\iota \to A)^E\,t :\equiv \lambda\,u.\,A^E\,(\mathsf{app}\,t\,u)$$

$$-^E : (\Gamma : \mathsf{Con})(\nu : \mathsf{Sub}\,\Omega\,\Gamma) \to \Gamma^S\,\mathsf{E}\,(\nu^A\,(\Omega^T\,\Omega\,\mathsf{id}))\,(\nu^D\,\omega^D)$$

$$\Gamma^E\,\nu\,x :\equiv A^E\,(\nu\,x)$$

In ι^E we use the same equation as in the definition of E. In $(\iota \to A)^E$ the definition is well-typed because of the same equation, but instantiated for the abstracted u term this time. All of this amounts to some additional path induction and transport fiddling in the (intensional) Agda formalization. We get induction for Ω as below.

$$\mathsf{Ind}_\Omega : (alg : \mathsf{DispAlg}\,\mathsf{TmAlg}_\Omega) \to \mathsf{Section}\,\mathsf{TmAlg}_\Omega\,alg$$

$$\mathsf{Ind}_\Omega\,(X^D, \omega^D) :\equiv (E, \Omega^E\,\Omega\,\mathsf{id})$$

2.4   Discussion

2.4.1   Comparison to F-algebras

A well-known alternative way for treating inductive types is to use certain cocontinuous endofunctors as a more semantic notion of signatures.

For example, single-sorted inductive types can be presented as endofunctors which preserve colimits of some ordinal-indexed chains. For instance, if we have an κ-cocontinuous $F : \mathbb{C} \to \mathbb{C}$, then algebras are given as $(X : |\mathbb{C}|)\times(\mathbb{C}(F\,X,\,X))$, morphisms as commuting squares, and Adámek's theorem [AK79] establishes the existence of initial algebras.

An advantage of this approach is that we can describe different classes of signatures by choosing different $\mathbb{C}$ categories:

- If $\mathbb{C}$ is **Set**, we get simple inductive types.

- If $\mathbb{C}$ is $\mathbf{Set}^I$ for some set I, we get indexed inductive types.

- If $\mathbb{C}$ is $\mathbf{Set}/I$, we get inductive-recursive types.

Another advantage of F-algebras is that signatures are a fairly semantic notion: they make sense even if we have no syntactic presentation at hand. That said, often we do need syntactic signatures, for use in proof assistants, or just to have a convenient notation for a class of cocontinuous functors.

An elegant way of carving out a large class of such functors is to consider polynomials as signatures. For example, when working in $\mathbf{Set}$, a signature is an element of $(S : \mathsf{Set}) \times (P : S \to \mathsf{Set})$, and (S, P) is interpreted as a functor as $X \mapsto (s : S) \times (P\, s \to X)$. The initial algebra is the W-type specified by S shapes and P positions. This yields infinitary inductive types as well.

However, it is not known how to get *inductive-inductive* signatures by picking the right $\mathbb{C}$ category and a functor. In an inductive-inductive signature, there may be multiple sorts, which can be indexed over previously declared sorts. For example, in the signature for categories we have $\mathsf{Obj} : \mathsf{Set}$ and $\mathsf{Mor} : \mathsf{Obj} \to \mathsf{Obj} \to \mathsf{Set}$, indexed twice over Obj. Some extensions are required to the idea of F-algebras:

- For inductive-inductive definitions with two sorts, Forsberg gives a specification with two functors, and a considerably more complex notion of algebras, involving dialgebras [NF13].

- For an arbitrary number of sorts, Altenkirch et al. [ACD$^+$18] use a "list" of functors, specified mutually with categories of algebras: each functor has as domain the semantic category of all previous sorts.

The functors-as-signatures approach gets significantly less convenient as we consider more general specifications. The approach of this thesis is the skip the middle ground between syntactic signatures and semantic categories of algebras: we treat syntactic signatures as a key component, and give direct semantic interpretation for them. Although we lose the semantic nature of F-algebras, our approach scales extremely well, all the way up to infinitary quotient-inductive-inductive types in Chapter 5, and to some extent to higher inductive-inductive types as well in Chapter 7.

If we look back at $-^A : \mathsf{Con} \to \mathsf{Set} \to \mathsf{Set}$, we may note that Γ^A yields a functor, in fact the same functor (up to isomorphism) that we would get from an F-algebra presentation. However, this is a coincidence in the single-sorted

case. With the F-algebra presentation, we can view $(X : |\mathbb{C}|) \times (\mathbb{C}(F\,X,\,X))$ as specifying the category of algebras as the total category of a displayed category (by viewing the Σ-type here as taking total categories; a Σ in **Cat**). In our approach, we aim to get the displayed categories directly, without talking about functors.

2.4.2   Generic Programming

Let's consider now our signatures and term algebras in the context of generic programming. This is largely future work, and we don't elaborate it much. But we can draw some preliminary conclusions and make some comparisons.

If a language can formalize inductive signatures and their semantics, that can be viewed as an implementation of generic programming over the described types. Compared to a purely mathematical motivation for this formalization, the requirements for practical generic programming are a bit more stringent.

- *Encoding overhead*: there should be an acceptable overhead in program size and performance when using generic representations. Size blowup can be an issue when writing proofs as well, when types and expressions become too large to mentally parse.

- *Strictness properties*: generic representations should compute as much as possible, ideally in exactly the same way as their non-generic counterparts.

Fixpoints of functors.   There is a sizable literature of using fixpoints of functors in generic programming, mainly in Haskell [Swi08, BH11, YHLJ09] and Agda [LM11, AAC+20]. We give a minimal example below for an Agda-like implementation.

We have an inductive syntax for some strictly positive functors, covering essentially the same signatures as Con.

$$
\begin{aligned}
&\mathsf{Sig} && : \mathsf{Set} \\
&\mathsf{Id} && : \mathsf{Sig} \\
&\mathsf{K}\top && : \mathsf{Sig} \\
&-\!\otimes\!- && : \mathsf{Sig} \to \mathsf{Sig} \to \mathsf{Sig} \\
&-\!\oplus\!- && : \mathsf{Sig} \to \mathsf{Sig} \to \mathsf{Sig}
\end{aligned}
$$

Id codes the identity functor, and $K\top$ codes the functor which is constantly $\top$. $-\otimes-$ and $-\oplus-$ are pointwise products and coproducts respectively. So we have the evident interpretation functions:

$$[\![-]\!] : \mathsf{Sig} \to \mathsf{Set} \to \mathsf{Set}$$
$$\mathsf{map} : (F : \mathsf{Sig}) \to (X_0 \to X_1) \to [\![F]\!]\, X_0 \to [\![F]\!]\, X_1$$

In Haskell and Agda, the easiest way to get initial algebras is to directly define the inductive fixpoint for each assumed $F : \mathsf{Sig}$:

$$\mathsf{Fix_F} : \mathsf{Set}$$
$$\mathsf{con_F} : [\![F]\!]\, \mathsf{Fix_F} \to \mathsf{Fix_F}$$

In Haskell, this definition is valid for arbitrary *semantic F* functor, because there is no termination checking or positivity checking. In Agda, the above definition is valid because the positivity checker is willing to look inside the definition of $[\![-]\!]$. However, Coq and Lean reject this definition. Next we establish weak initiality, by defining the recursor:

$$\mathsf{rec} : ([\![F]\!]\, X \to X) \to \mathsf{Fix_F} \to X$$
$$\mathsf{rec}\, f\, (\mathsf{con}\, x) :\equiv f\, (\mathsf{map}\, (\mathsf{rec}\, f)\, x)$$

This is again fine in Haskell, but it unfortunately does not pass Agda's termination checker. The most conservative solution is to inline the recursive call into map, so that the definition becomes transparent to termination checking.

$$\mathsf{rec} : ([\![F]\!]\, X \to X) \to \mathsf{Fix_F} \to X$$
$$\mathsf{rec}\, f\, (\mathsf{con}\, x) :\equiv f\, (\mathsf{maprec}\, f\, x)$$

$$\mathsf{maprec} : \{G\} \to ([\![F]\!]\, X \to X) \to [\![G]\!]\, \mathsf{Fix_F} \to [\![G]\!]\, X$$
$$\mathsf{maprec}\, \{\mathsf{Id}\} \qquad f\, x \qquad :\equiv \mathsf{rec}\, f\, x$$
$$\mathsf{maprec}\, \{\mathsf{K\top}\} \qquad f\, x \qquad :\equiv x$$
$$\mathsf{maprec}\, \{G \otimes H\}\, f\, (x,\, y) \quad :\equiv (\mathsf{maprec}\, f\, x,\, \mathsf{maprec}\, f\, y)$$
$$\mathsf{maprec}\, \{G \oplus H\}\, f\, (\mathsf{inj}_1\, x) :\equiv \mathsf{maprec}\, f\, x$$
$$\mathsf{maprec}\, \{G \oplus H\}\, f\, (\mathsf{inj}_2\, x) :\equiv \mathsf{maprec}\, f\, x$$

Alternatively, we might use *sized types* [AVW17], as in [AAC$^+$20]. The drawback is dependence on an additional language feature which is only supported in Agda, and which has had several soundness issues since its introduction [**?**].

There is yet another possible approach: defining initial algebras as sequential colimits, using Adámek's theorem. This approach was taken by Ahrens, Matthes and Mörtberg in [AMM19]. However, the encoding overhead is excessive, and it is practically unusable for generic programming. Another drawback is that defining colimits requires quotient types, which are often not available natively.

W-types.   Given a polynomial $(S,\, P) : (S : \mathsf{Set}) \times (S \to \mathsf{Set})$, the corresponding W-type is inductively specified as below:

$$\mathsf{W}_{S,\, P} : \mathsf{Set}$$
$$\mathsf{sup} \quad : (s : S) \to (P\, s \to \mathsf{W}_{S,\, P}) \to \mathsf{W}_{S,\, P}$$

If we assume $\top$, $\bot$, Bool, Π, Σ, W-types, universes and an intensional identity type, a large class of inductive types can be derived, including infinitary and finitary indexed inductive families; this was shown by Hugunin [Hug21]. The encoding also yields definitional β-rules for recursion and elimination. However, there is also significant encoding overhead here.

- First, there is a translation from more convenient signatures to $(S,\, P)$ polynomials.

- Then, we take the $\mathsf{W}_{S\, P}$ type, but we need to additionally restrict it to the *canonical* elements by a predicate, as in $(x : \mathsf{W}_{S\, P}) \times \mathsf{canonical}\, x$. This is required because the only way to represent inductive branching is by functions, but functions sometimes contain too many elements up to definitional equality. For example, $\bot \to A$ has infinitely many definitonally distinct inhabitants.

There is also a performance overhead imposed by the mandatory higher-order constructors. W-types are a great way to have a small basis in a formal setting, both in intensional and extensional type theories, but they are a bit too heavy for practical purposes.

Term algebras.   The main advantage of the term algebras described in this chapter is that they can be defined using plain inductive families, which are available in every major dependently typed language. Even in GHC Haskell, the support for generalized ADTs is sufficient for implementing term algebras. In contrast, as we have seen, the direct inductive definition of fixpoints is only

possible in Agda, while the colimit definition requires quotients and is rather heavyweight. Even in Agda, the drawback of the direct fixpoint definition is that many generic operations only pass termination checking after manual inlining. With term algebras, generic operations can be defined by induction on $\mathsf{Tm}\,\Gamma\,A$ in an obviously terminating way.

For practical usage it makes sense to slightly modify terms. We switch to a *spine neutral* definition. We mutually inductively define Spine and Tm:

$$
\begin{aligned}
&\mathsf{Spine} : (\Gamma : \mathsf{Con}) \to \mathsf{Ty} \to \mathsf{Ty} \to \mathsf{Set} \\
&\epsilon \quad\;\; : \{A\} \to \mathsf{Spine}\,\Gamma\,A\,A \\
&\text{--},\text{--} \;\; : \{B\,C\} \to \mathsf{Tm}\,\Gamma\,\iota \to \mathsf{Spine}\,\Gamma\,B\,C \to \mathsf{Spine}\,\Gamma\,(\iota \to B)\,C \\
&\mathsf{Tm} \quad : \mathsf{Con} \to \mathsf{Ty} \to \mathsf{Set} \\
&\text{--}\$\text{--} \;\; : \{A\,B\} \to \mathsf{Var}\,\Gamma\,A \to \mathsf{Spine}\,\Gamma\,A\,B \to \mathsf{Tm}\,\Gamma\,B
\end{aligned}
$$

With this representation, a term is always a variable applied to a list of arguments. This can be useful, because pattern matching implementations in metalanguages (e.g. Agda or Idris) are more likely to know about which constructors are possible in patterns. Using this, we give here an ad-hoc definition of natural numbers in pseudo-Agda.

$$
\begin{aligned}
&\mathsf{Nat} : \mathsf{Set} && \mathsf{zero} \;\; :\equiv \mathsf{vz}\,\$\,\epsilon \\
&\mathsf{Nat} :\equiv \mathsf{Tm}\,(\bullet \rhd \iota \to \iota \rhd \iota)\,\iota && \mathsf{suc}\,n :\equiv \mathsf{vs}\,\mathsf{vz}\,\$\,(n,\,\epsilon)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{NatElim} : (P : \mathsf{Nat} \to \mathsf{Set}) \to P\,\mathsf{zero} \to ((n : \mathsf{Nat}) \to P\,n \to P\,(\mathsf{suc}\,n)) \\
&\qquad\qquad\;\; \to (n : \mathsf{Nat}) \to P\,n \\
&\mathsf{NatElim}\,P\,\mathsf{pz}\,\mathsf{ps}\,(\mathsf{vz}\,\$\,\epsilon) \qquad\quad :\equiv \mathsf{pz} \\
&\mathsf{NatElim}\,P\,\mathsf{pz}\,\mathsf{ps}\,(\mathsf{vs}\,\mathsf{vz}\,\$\,(n,\,\epsilon)) :\equiv \mathsf{ps}\,n\,(\mathsf{NatElim}\,P\,\mathsf{pz}\,\mathsf{ps}\,n)
\end{aligned}
$$

The actual Agda definition can be found in the supplementary formalization, and it is pretty much the same as above. We recover the exact same behavior with respect to pattern matching as with native inductive definitions.

Rec_Ω and Ind_Ω can be adapted to spine neutral terms with minor adjustments. But what about the β-rules which they produce as part of their output, are they definitional (i.e. proven by refl)? In this chapter we do not have a rigorous way of reasoning about definitional equalities; in the next chapter we develop such

reasoning and show that Rec_Ω enjoys definitional β-rules (with or without the spine neutral definition).

However, Ind_Ω only supports propositional β-rules. The issue is the extra transporting in the definition of $\mathsf{E} : (t : \mathsf{Tm}\,\Omega\,\iota) \to X^D\,t$: transports don't strictly commute with constructors. This appears to be a point of advantage for the direct fixpoint definition in Agda, as it allows generic elimination with strict computation rules [?].

The term algebra presentation can be easily extended to indexed families. In that case, signatures and terms are still definable with basic inductive families, without requiring quotients or complicated encodings; see Kaposi and von Raumer [KvR20].

A related existing work is *sum-of-products* generics by De Vries and Löh [dVL14]. There, signatures for functors are in a normal form: we cannot freely take products and coproducts, instead a signature looks very much like a Con in this chapter (except in an indexed form). The authors observe that several generic programming patterns are easier to express with normalized signatures. However, they still use explicit fixpoints as the way to get initial algebras.

CHAPTER 3

Semantics in Two-Level Type Theory

In this chapter we describe how two-level type theory is used as a metatheoretic setting in the rest of this thesis. First, we provide motivation and overview. Second, we describe models of type theories in general, and models of two-level type theories as extensions. Third, we describe presheaf models of two-level type theories. Finally, we generalize the semantics and the term algebra construction from Chapter 2 in two-level type theory, as a way to illustrate the applications.

3.1 Motivation

We note two shortcomings of the semantics presented in the previous chapter.

First, the semantics that we provided was not as general as it could be. We used the internal Set universe to specify algebras, but algebras make sense in many different categories. A crude way to generalize semantics is to simply say that our formalization, which was carried out in the syntax (i.e. initial model) of some intensional type theory, can be interpreted in any model of the type theory. However, this is wasteful: for simple inductive signatures, it is enough to assume a category with finite products as semantic setting. We don't need all the extra baggage that comes with a model of a type theory.

Second, we were not able to reason about definitional equalities, only propositional ones. We have a formalization of signatures and semantics in intensional Agda, where the two notions differ[1], but only propositional equality is subject to internal reasoning. For instance, we would like to show that term algebras support recursion with strict β-rules, and for this we need to reason about strict

---

[1]As opposed to in extensional type theory, where they're the same.

equality.

Notation 3. We use $\bullet$ for the terminal object in a $\mathbb{C}$ category, with $\epsilon : \mathbb{C}(A, \bullet)$ for the unique morphism. For products, we use $- \otimes -$ with $(\text{-},\text{-}) : \mathbb{C}(A, B) \to \mathbb{C}(A, C) \to \mathbb{C}(A, B \otimes C)$ and p and q for first and second projections respectively.

Example 6. Assuming $\mathbb{C}$ has finite products, natural number algebras and binary tree algebras are specified as follows.

$$\mathsf{Alg}_{\mathsf{NatSig}} :\equiv (X : |\mathbb{C}|) \times \mathbb{C}(\bullet, X) \times \mathbb{C}(X, X)$$
$$\mathsf{Alg}_{\mathsf{TreeSig}} :\equiv (X : |\mathbb{C}|) \times \mathbb{C}(\bullet, X) \times \mathbb{C}(X \otimes X, X)$$

Here, $\mathsf{Alg}_{\mathsf{NatSig}}$ and $\mathsf{Alg}_{\mathsf{TreeSig}}$ are both sets in some metatheory, and the $\times$ in the definitions refer to the metatheoretic Σ. Algebras can be viewed as diagrams which preserve finite products, and algebra morphisms are natural transformations.

How should we adjust Alg to compute algebras in $\mathbb{C}$, and Mor to compute their morphisms? While it is possible to do this in a direct fashion, working directly with objects and morphisms of $\mathbb{C}$ is rather unwieldy. $\mathbb{C}$ is missing many convenience features of type theories.

- There are no variables or binders. We are forced to work in a point-free style or chase diagrams; both become difficult to write and read after a certain point of complexity.

- There no functions, universes or inductive types.

- Substitution (with weakening as a special case) has to be handled explicitly and manually. Substitutions are certain morphisms, while "terms" are also morphisms, and we have to use composition to substitute terms. In contrast, if we are working internally in a type theory, terms and substitutions are distinct, and we only have to explicitly deal with terms, and substitutions are automated and implicit.

The above overlaps with motivations for working in *internal languages* [nc21] of structured categories: they aid calculation and compact formalization by hiding bureaucratic structural details.

A finite product category $\mathbb{C}$ does not have much of an internal language, it is too bare-bones. But we can work instead in the internal language of $\hat{\mathbb{C}}$, the

category of presheaves over $\mathbb{C}$. This allows faithful reasoning about $\mathbb{C}$, while also including all convenience features of extensional type theory.

Two-level type theories [ACKS19], or 2LTT in short, are type theories such that they have "standard" interpretations in presheaf categories. A 2LTT has an inner layer, where types and terms arise by embedding $\mathbb{C}$ in $\hat{\mathbb{C}}$, and an outer layer, where constructions are inherited from $\hat{\mathbb{C}}$. The exact details of the syntax may vary depending on what structures $\mathbb{C}$ supports, and which type formers we assume in the outer layer. Although it is possible to add assumptions to a 2LTT which preclude standard presheaf semantics [ACKS19, Section 2.4.], we stick to basic 2LTT in this thesis. By using 2LTT, we are able to use a type-theoretic syntax in the rest of the thesis which differs only modestly from the style of definitions that we have seen so far.

From a programming perspective, basic 2LTT provides a convenient syntax for writing metaprograms. This can be viewed as *two-stage compilation*: if we have a 2LTT program with an inner type, we can run it, and it returns another program, which lives purely in the inner theory.

3.2 Models of Type Theories

Before elaborating on 2LTT-specific features, we review models of type theories in general. Variants of 2LTT will be obtained by adding extra features on the top of more conventional TTs.

It is also worth to take a more general look at models at this point, because the notions presented in this subsection (categories with families, type formers) will be reused several times in this thesis, when specifying theories of signatures.

3.2.1 The Algebraic View

We take an algebraic view [**?**] of models and syntaxes of type theories throughout this thesis. Models of type theories are algebraic structures: they are categories with certain extra structure. The syntax of a type theory is understood to be its initial model. In initial models, the underlying category is the category of typing contexts and parallel substitutions, while the extra structure corresponds to type and term formers, and equations quotient the syntax by definitional equality.

Type theories can be described with quotient inductive-inductive (QII) signatures, and their initial models are quotient inductive-inductive types (QIITs).

Hence, 2LTT is also a QII theory. We will first talk about QIITs in Chapter 4. Until then, we shall make do with an informal understanding of categorical semantics for type theories, without using anything in particular from the metatheory of QIITs. There is some annoying circularity here, that we talk about QIITs in this thesis, but we employ QIITs when talking about them. However, this is only an annoyance in exposition and not a fundamental issue: Chapter 6 describes how to eliminate circularity by a form of bootstrapping.

The algebraic view lets us dispense with all kinds of "raw" syntactic objects. We only ever talk about well-typed and well-formed objects, moreover, every construction must respect definitional equalities. For terms in the algebraic syntax, definitional equality coincides with metatheoretic equality. This mirrors equality of morphisms in 1-category theory, where we usually reuse metatheoretic equality.

In the following we specify notions of models for type theories. We split this in two parts: categories with families and type formers.

3.2.2   Categories With Families

Definition 12. A **category with family** (cwf) [Dyb95] is a way to specify the basic structural rules for contexts, substitutions, types and terms. It yields a dependently typed explicit substitution calculus [**?**]. A cwf consists of the following.

- A category with a terminal object. We denote the set of objects as $\mathsf{Con} : \mathsf{Set}$ and use capital Greek letters starting from Γ to refer to objects. The set of morphisms is $\mathsf{Sub} : \mathsf{Con} \to \mathsf{Con} \to \mathsf{Set}$, and we use σ, δ and so on to refer to morphisms. We write id for the identity morphism and $- \circ -$ for composition. The terminal object is $\bullet$ with unique morphism $\epsilon : \mathsf{Sub}\,\Gamma\,\bullet$. In initial models (that is, syntaxes) of type theories, objects correspond to typing contexts, morphisms to parallel substitutions and the terminal object to the empty context; this informs the naming scheme.

- A *family structure*, containing $\mathsf{Ty} : \mathsf{Con} \to \mathsf{Set}$ and $\mathsf{Tm} : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Set}$. We use A, B, C to refer to types and t, u, v to refer to terms. Ty is a presheaf over the category of contexts and Tm is a displayed presheaf

over Ty. This means that types and terms can be substituted:

$$-[-] : \mathsf{Ty}\,\Delta \to \mathsf{Sub}\,\Gamma\,\Delta \to \mathsf{Ty}\,\Gamma$$
$$-[-] : \mathsf{Tm}\,\Delta\,A \to (\sigma : \mathsf{Sub}\,\Gamma\,\Delta) \to \mathsf{Tm}\,\Gamma\,(A[\sigma])$$

Substitution is functorial: we have $A[\mathsf{id}] = A$ and $A[\sigma \circ \delta] = A[\sigma][\delta]$, and likewise for terms.

A family structure is additionally equipped with *context comprehension* which consists of a context extension operation $-\triangleright- : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Con}$ together with an isomorphism $\mathsf{Sub}\,\Gamma\,(\Delta \triangleright A) \simeq ((\sigma : \mathsf{Sub}\,\Gamma\,\Delta) \times \mathsf{Tm}\,\Gamma\,(A[\sigma]))$ which is natural in Γ.

The following notions are derivable from the comprehension structure:

- By going right-to-left along the isomorphism, we recover *substitution extension* $-,- : (\sigma : \mathsf{Sub}\,\Gamma\,\Delta) \to \mathsf{Tm}\,\Gamma\,(A[\sigma]) \to \mathsf{Sub}\,\Gamma\,(\Delta \triangleright A)$. This means that starting from ϵ or the identity substitution id, we can iterate $-,-$ to build substitutions as lists of terms.

- By going left-to-right, and starting from $\mathsf{id} : \mathsf{Sub}\,(\Gamma \triangleright A)\,(\Gamma \triangleright A)$, we recover the *weakening substitution* $\mathsf{p} : \mathsf{Sub}\,(\Gamma \triangleright A)\,\Gamma$ and the *zero variable* $\mathsf{q} : \mathsf{Tm}\,(\Gamma \triangleright A)\,(A[\mathsf{p}])$.

- By weakening q, we recover a notion of variables as De Bruijn indices. In general, the n-th De Bruijn index is defined as $\mathsf{q}[\mathsf{p}^n]$, where p^n denotes n-fold composition.

Comprehension can be characterized either by taking $-,-$, p and q as primitive, or the natural isomorphism. The two are equivalent, and we may switch between them, depending on which is more convenient.

There are other ways for presenting the basic categorical structure of models, which are nonetheless equivalent to cwfs, including natural models [Awo18] and categories with attributes [Car78]. We use the cwf presentation for its immediately algebraic character and closeness to conventional explicit substitution syntax.

Notation 4. As De Bruijn indices are hard to read, we will mostly use nameful notation for binders. For example, assuming $\mathsf{Nat} : \{\Gamma : \mathsf{Con}\} \to \mathsf{Ty}\,\Gamma$ and $\mathsf{Id} : \{\Gamma : \mathsf{Con}\}(A : \mathsf{Ty}\,\Gamma) \to \mathsf{Tm}\,\Gamma\,A \to \mathsf{Tm}\,\Gamma\,A \to \mathsf{Ty}\,\Gamma$, we may write $\bullet \triangleright n : \mathsf{Nat} \triangleright$

p : Id Nat $n\,n$ for a typing context, instead of using numbered variables or cwf combinators as in $\bullet \triangleright$ Nat $\triangleright$ Id Nat q q.

Notation 5. In the following, we will denote family structures by (Ty,Tm) pairs and overload context extension $- \triangleright -$ for different families.

Definition 13. The following derivable operations are commonly used.

- *Single substitution* can be derived from parallel substitution as follows. Assume t : Tm $(\Gamma \triangleright A)\,B$, and u : Tm $\Gamma\,A$. t is a term which may depend on the last variable in the context, which has A type. We can substitute that variable with the u term as $t[\mathsf{id}, u]$: Tm $\Gamma\,(A[\mathsf{id}, u])$. Note that term substitution causes the type to be substituted as well. (id, u) : Sub $\Gamma\,(\Gamma \triangleright A)$ is well-typed because u : Tm $\Gamma\,A$ hence also u : Tm $\Gamma\,(A[\mathsf{id}])$.

- We can *lift substitutions* over binders as follows. Assuming σ : Sub $\Gamma\,\Delta$ and A : Ty Δ, we construct a lifting of σ which maps an additional A-variable to itself: $(\sigma \circ \mathsf{p}, \mathsf{q})$: Sub $(\Gamma \triangleright A[\sigma])\,(\Delta \triangleright A)$. Let us see why this is well-typed. We have p : Sub $(\Gamma \triangleright A[\sigma])\,\Gamma$ and σ : Sub $\Gamma\,\Delta$, so $\sigma \circ \mathsf{p}$: Sub $(\Gamma \triangleright A[\sigma])\,\Delta$. Also, q : Tm $(\Gamma \triangleright A[\sigma])\,(A[\sigma][\mathsf{p}])$, hence q : Tm $(\Gamma \triangleright A[\sigma])\,(A[\sigma \circ \mathsf{p}])$, thus $(\sigma \circ \mathsf{p}, \mathsf{q})$ typechecks.

Notation 6. As a nameful notation for substitutions, we may write $t[x \mapsto u]$, for a single substitution, or $t[x \mapsto u_1, y \mapsto u_2]$ and so on.

In nameful notation we leave all weakening implicit, including substitution lifting. Formally, if we have t : Tm $\Gamma\,A$, we can only mention t in Γ. If we need to mention it in $\Gamma \triangleright B$, we need to use $t[\mathsf{p}]$ instead. In the nameful notation, t : Tm $(\Gamma \triangleright x : B)\,A$ may be used.[2]

3.2.3   Type formers

A family structure in a cwf may be closed under certain type formers, such as functions, Σ-types, universes or inductive types. We give some examples here for their specification. First, we look at common negative type formers, which can be specified using isomorphisms. Then, we consider positive type formers, and finally universes.

---

[2]Moreover, when working in the internal syntax of a theory, we just write Agda-like type-theoretic notation, without noting contexts and substitutions in any way.

Negative types

Definition 14. A $(\mathsf{Ty}, \mathsf{Tm})$ family supports Π**-types** if it supports the following.

$$
\begin{aligned}
\Pi \quad &: (A : \mathsf{Ty}\,\Gamma) \to \mathsf{Ty}\,(\Gamma \triangleright A) \to \mathsf{Ty}\,\Gamma \\
\Pi[] \quad &: (\Pi\,A\,B)[\sigma] = \Pi\,(A[\sigma])\,(B[\sigma \circ \mathsf{p}, \mathsf{q}]) \\
\mathsf{app} \quad &: \mathsf{Tm}\,\Gamma\,(\Pi\,A\,B) \to \mathsf{Tm}\,(\Gamma \triangleright A)\,B \\
\mathsf{lam} \quad &: \mathsf{Tm}\,(\Gamma \triangleright A)\,B \to \mathsf{Tm}\,\Gamma\,(\Pi\,A\,B) \\
\Pi\beta \quad &: \mathsf{app}\,(\mathsf{lam}\,t) = t \\
\Pi\eta \quad &: \mathsf{lam}\,(\mathsf{app}\,t) = t \\
\mathsf{lam}[] &: (\mathsf{lam}\,t)[\sigma] = \mathsf{lam}\,(t[\sigma \circ \mathsf{p}, \mathsf{q}])
\end{aligned}
$$

Here, Π is the type formation rule. $\Pi[]$ is the type substitution rule, expressing that substituting Π proceeds structurally on constituent types. Note $B[\sigma \circ \mathsf{p}, \mathsf{q}]$, where we lift σ over the additional binder.

The rest of the rules specify a natural isomorphism $\mathsf{Tm}\,\Gamma\,(\Pi\,A\,B) \simeq \mathsf{Tm}\,(\Gamma \triangleright A)\,B$. We only need a substitution rule (i.e. a naturality rule) for one direction of the isomorphism, since the naturality of the other map is derivable.

This way of specifying Π-types is very convenient if we have explicit substitutions. The usual "pointful" specification is equivalent to this. For example, we have the following derivation of pointful application:

$$
\begin{aligned}
\mathsf{app}' &: \mathsf{Tm}\,\Gamma\,(\Pi\,A\,B) \to (u : \mathsf{Tm}\,\Gamma\,A) \to \mathsf{Tm}\,\Gamma\,(B[\mathsf{id}, u]) \\
\mathsf{app}'\,t\,u &:\equiv (\mathsf{app}\,t)[\mathsf{id}, u]
\end{aligned}
$$

Remark on naturality. The above specification for Π can be written more compactly if we assume that everything is natural with respect to substitution.

$$
\begin{aligned}
\Pi \quad &: (A : \mathsf{Ty}\,\Gamma) \to \mathsf{Ty}\,(\Gamma \triangleright A) \to \mathsf{Ty}\,\Gamma \\
(\mathsf{app}, \mathsf{lam}) &: \mathsf{Tm}\,\Gamma\,(\Pi\,A\,B) \simeq \mathsf{Tm}\,(\Gamma \triangleright A)\,B
\end{aligned}
$$

This is a reasonable assumption; in the rest of the thesis we only ever define structures on cwfs which are natural in this way.

Notation 7. From now on, when specifying type formers in family structures, we assume that everything is natural, and thus omit substitution equations.

There are ways to make this idea more precise, and take it a step further by working in languages where only natural constructions are possible. The term

higher-order abstract syntax (HOAS) is sometimes used for this style. It lets us also omit contexts, so we would only need to write

$$\Pi \qquad\qquad : (A : \mathsf{Ty}) \to (\mathsf{Tm}\,A \to \mathsf{Ty}) \to \mathsf{Ty}$$
$$(\mathsf{app}, \mathsf{lam}) : \mathsf{Tm}\,(\Pi\,A\,B) \simeq ((a : \mathsf{Tm}\,A) \to \mathsf{Tm}\,(B\,a))$$

Recently several promising works emerged in this area [**?**]. Although this technology is likely to be the preferred future direction in the metatheory of type theories, this thesis does not make use of it. The field is rather fresh, with several different approaches and limited amount of pedagogical exposition, and the new techniques would also raise the level of abstraction in this thesis, all contributing to making it less accessible. It's also not obvious how exactly HOAS-style could be employed to aid formalization here, and it would require significant additional research. Often, a setup with multiple modalities ("multimodal" [GKNB20]) is required [BKS21], because we work with presheaves over different cwfs. It seems that a synthetic notion of dependent modes would be also required to formalize constructions in this thesis, since we often work with displayed presheaves over displayed cwfs. This is however not yet developed in literature.

Definition 15. A family structure supports **constant families** if we have the following.

$$\mathsf{K} \qquad\qquad : \mathsf{Con} \to \{\Gamma : \mathsf{Con}\} \to \mathsf{Ty}\,\Gamma$$
$$(\mathsf{app}_\mathsf{K}, \mathsf{lam}_\mathsf{K}) : \mathsf{Tm}\,\Gamma\,(\mathsf{K}\,\Delta) \simeq \mathsf{Sub}\,\Gamma\,\Delta$$

Constant families express that every context can be viewed as a non-dependent type in any context. Having constant families is equivalent to the *democracy* property for a cwf [CD14, NF13]. Constant families are convenient when building models, because they let us model non-dependent types as semantic contexts, which are often simpler structures than semantic types. From a programming perspective, constant families specify closed record types, where $\mathsf{K}\,\Delta$ has Δ-many fields.

If we have equalities of sets for the specification, i.e. $\mathsf{Tm}\,\Gamma\,(\mathsf{K}\,\Delta) = \mathsf{Sub}\,\Gamma\,\Delta$, we have **strict constant families**.

Definition 16. A family structure supports Σ**-types** if we have

$$\Sigma \qquad\qquad : (A : \mathsf{Ty}\,\Gamma) \to \mathsf{Ty}\,(\Gamma \triangleright A) \to \mathsf{Ty}\,\Gamma$$
$$(\mathsf{proj}, (-, -)) : \mathsf{Tm}\,\Gamma\,(\Sigma\,A\,B) \simeq ((t : \mathsf{Tm}\,\Gamma\,A) \times \mathsf{Tm}\,\Gamma\,(B[\mathsf{id}, t]))$$

We use the shorter specification above, where everything is assumed to be natural. We may write proj_1 and proj_2 for composing the metatheoretic first and second projections with proj.

Definition 17. A family structure supports the **unit type** if we have $\top : \mathsf{Ty}\,\Gamma$ such that $\mathsf{Tm}\,\Gamma\,\top \simeq \top$, where the $\top$ on the right is the metatheoretic unit type, and we overload $\top$ for the internal unit type. From this, we get the internal $\mathsf{tt} : \mathsf{Tm}\,\Gamma\,\top$, which is definitionally unique.

Definition 18. A family structure supports **extensional identity** types if there is $\mathsf{Id} : \mathsf{Tm}\,\Gamma\,A \to \mathsf{Tm}\,\Gamma\,A \to \mathsf{Ty}\,\Gamma$ such that $(\mathsf{reflect}, \mathsf{refl}) : \mathsf{Tm}\,\Gamma\,(\mathsf{Id}\,t\,u) \simeq (t = u)$.

It is also possible to give a positive definition for identity types, in which case we get intensional identity. Extensional identity corresponds to a categorical equalizer of terms (a limit), while the Martin-Löf-style intensional identity is characterized as the initial reflexive relation on a type (a colimit).

This choice between negative and positive specification generally exists for type formers with a single term construction rule. For example, Σ can be defined as a positive type, with an elimination rule that behaves like pattern matching. Positive Σ is equivalent to negative Σ, although it only supports propositional η-rules. In contrast, positive identity is usually *not* equivalent to negative identity.

$\mathsf{refl} : t = u \to \mathsf{Tm}\,\Gamma\,(\mathsf{Id}\,t\,u)$ expresses reflexivity of identity: definitionally equal terms are provably equal. $\mathsf{reflect}$, which goes the other way around, is called *equality reflection* [**?**]: provably equal terms are identified in the metatheory.

Uniqueness of identity proofs (UIP) is often ascribed to the extensional identity type [**?**]. UIP means that $\mathsf{Tm}\,\Gamma\,(\mathsf{Id}\,t\,u)$ has at most a single inhabitant up to Id. However, UIP is not something which is inherent in the negative specification, instead it's inherited from the metatheory. If Tm forms a homotopy set in the metatheory, then internal equality proofs inherit uniqueness through the defining isomorphism.

Positive types

We do not dwell much on positive types here, as elsewhere in this thesis we talk a lot about specifying such types anyway. We provide here some background and a small example.

The motivation is to specify initial internal algebras in a cwf. However, specifying the uniqueness of recursors using definitional equality is problematic, if we

are to have decidable and efficient conversion checking for a type theory. Consider the specification of Bool together with its recursor.

$$
\begin{aligned}
&\mathsf{Bool} &&: \mathsf{Ty}\,\Gamma \\
&\mathsf{true} &&: \mathsf{Tm}\,\Gamma\,\mathsf{Bool} \\
&\mathsf{false} &&: \mathsf{Tm}\,\Gamma\,\mathsf{Bool} \\
&\mathsf{BoolRec} &&: (B : \mathsf{Ty}\,\Gamma) \to \mathsf{Tm}\,\Gamma\,B \to \mathsf{Tm}\,\Gamma\,B \to \mathsf{Tm}\,\Gamma\,\mathsf{Bool} \to \mathsf{Tm}\,\Gamma\,B \\
&\mathsf{true}\beta &&: \mathsf{BoolRec}\,B\,t\,f\,\mathsf{true} = t \\
&\mathsf{false}\beta &&: \mathsf{BoolRec}\,B\,t\,f\,\mathsf{false} = f
\end{aligned}
$$

BoolRec together with the β-rules specifies an internal Bool-algebra morphism. A possible way to specify definitional uniqueness is as follows. Assuming $B : \mathsf{Ty}\,\Gamma$, $t : \mathsf{Tm}\,\Gamma\,B$, $f : \mathsf{Tm}\,\Gamma\,B$ and $m : \mathsf{Tm}\,(\Gamma \rhd b : \mathsf{Bool})\,B$, such that $m[b \mapsto \mathsf{true}] = t$ and $m[b \mapsto \mathsf{false}] = f$, it follows that $\mathsf{BoolRec}\,B\,t\,f\,b : \mathsf{Tm}\,(\Gamma \rhd b : \mathsf{Bool})\,B$ is equal to m.

Unfortunately, deciding conversion with this rule entails deciding pointwise equality of arbitrary Bool functions, which can be done in exponential time in the number of Bool arguments. More generally, Scherer presented a decision algorithm for conversion checking with strong finite sums and products in simple type theory [Sch17], which also takes exponential time. If we move to natural numbers with definitionally unique recursion, conversion checking becomes undecidable.

One solution is to have propositionally unique recursion instead. However, if such equations are postulated, that would break the canonicity property in intensional type theories, since now we would have equality proofs other than refl in the empty context.

The standard solution is to have dependent elimination principles instead: this allows inductive reasoning, canonicity and effectively decidable definitional equality at the same time. For Bool, we would have

$$
\begin{aligned}
\mathsf{BoolInd} : {}&(B : \mathsf{Ty}\,(\Gamma \rhd b : \mathsf{Bool})) \to \mathsf{Tm}\,\Gamma\,(B[b \mapsto \mathsf{true}]) \\
&\to \mathsf{Tm}\,\Gamma\,(B[b \mapsto \mathsf{false}]) \to (t : \mathsf{Tm}\,\Gamma\,\mathsf{Bool}) \to \mathsf{Tm}\,\Gamma\,(B[b \mapsto t])
\end{aligned}
$$

together with $\mathsf{BoolInd}\,B\,t\,f\,\mathsf{true} = t$ and $\mathsf{BoolInd}\,B\,t\,f\,\mathsf{false} = f$.

Of course, if we assume extensional identity types, we have undecidable conversion anyway, and definitionally unique recursion is equivalent to induction. But decidable conversion is an essential part of type theory, perhaps its main

selling point as a foundation for mechanized mathematics, which makes it possible to relegate a deluge of boilerplate to computers. Hence, decidable conversion should be kept in mind.

Universes

Universes are types which classify types. There are several different flavors of universes.

Definition 19. A **Tarski-style** universe consists of the following data:

$$\mathsf{U} : \mathsf{Ty}\,\Gamma \qquad \mathsf{El} : \mathsf{Tm}\,\Gamma\,\mathsf{U} \to \mathsf{Ty}\,\Gamma$$

This is a weak classifier, since not all $\mathsf{Ty}\,\Gamma$ are necessarily represented as terms of the universe. Rather, this kind of universe can be viewed as an internal subfamily of $(\mathsf{Ty}, \mathsf{Tm})$. Like families, Tarski universes can be closed under type formers. For instance, if U has Nat, we have the following:

$$\mathsf{Nat} : \mathsf{Tm}\,\Gamma\,\mathsf{U} \quad zero : \mathsf{Tm}\,\Gamma\,(\mathsf{El}\,\mathsf{Nat}) \quad suc : \mathsf{Tm}\,\Gamma\,(\mathsf{El}\,\mathsf{Nat}) \to \mathsf{Tm}\,\Gamma\,(\mathsf{El}\,\mathsf{Nat})$$

$$\begin{aligned} \mathsf{NatElim} : \quad &(P : \mathsf{Ty}\,(\Gamma \triangleright n : \mathsf{El}\,\mathsf{Nat})) \\ &\to \mathsf{Tm}\,\Gamma\,(P[n \mapsto zero]) \\ &\to \mathsf{Tm}\,(\Gamma \triangleright n : \mathsf{El}\,\mathsf{Nat} \triangleright np : P[n \mapsto n])\,(P[n \mapsto suc\,n]) \\ &\to (n : \mathsf{Tm}\,\Gamma\,(\mathsf{El}\,\mathsf{Nat})) \to \mathsf{Tm}\,\Gamma\,(P[n \mapsto n]) \end{aligned}$$

If all type formers in U follow this scheme, U may be called a **weakly Tarski** universe. If we assume that every type former in U is also duplicated in $(\mathsf{Ty}, \mathsf{Tm})$, moreover El preserves all type formers, so that e.g. $\mathsf{El}\,\mathsf{Nat}$ is definitionally equal to the natural number type in Ty, then U is **strongly Tarski**.

It is often more convenient to have stronger classifiers as universes, so that *all* types in a given family structure are represented.

Definition 20. Ignoring size issues for now, **Coquand universes** [?] are specified as follows:

$$\mathsf{U} : \mathsf{Ty}\,\Gamma \quad (\mathsf{El}, \mathsf{c}) : \mathsf{Tm}\,\Gamma\,\mathsf{U} \simeq \mathsf{Ty}\,\Gamma$$

c maps every type in Ty to a code in U. Now we can ignore El when specifying type formers, as c can be always used to get a code in U for a type.

Unfortunately, the exact specification above yields an inconsistent "type-in-type" system, because U itself has a code in U. The standard solution is to have multiple family structures $(\mathsf{Ty}_i, \mathsf{Tm}_i)$, indexed by universe levels, and have $U_i : \mathsf{Ty}_{i+1}\,\Gamma$ and $\mathsf{Tm}_{i+1}\,\Gamma\,U_i \simeq \mathsf{Ty}_i\,\Gamma$. For a general specification of consistent universe hierarchies , see [Kov21]. As mentioned in Section 1.3.2, we omit universe indices in the following, and implicitly assume "just enough" universes for particular purposes.

Definition 21. Russell universes are Coquand universes additionally satisfying $\mathsf{Tm}\,\Gamma\,U = \mathsf{Ty}\,\Gamma$ as an equality of sets, and also $\mathsf{El}\,t = t$. This justifies omitting El and c from informal notation, implicitly casting between $\mathsf{Tm}\,\Gamma\,U$ and $\mathsf{Ty}\,\Gamma$.

Russell-style universes are commonly supported in set-theoretic models. They are also often inherited from meta-type-theories which themselves have Russell-universes. Major implementations of type theories (Coq, Lean, Agda, Idris) are all such.

3.3   Two-Level Type Theory

3.3.1   Models

We describe models of 2LTT in the following. This is not the only possible way to present 2LTT; our approach differs from [ACKS19] in some ways. We will summarize the differences at the end of this section.

Definition 22. A model of a **two-level type theory** is a model of type theory such that

- It supports a Tarski-style universe $\mathsf{Ty}_0 : \mathsf{Ty}\,\Gamma$ with decoding $\mathsf{Tm}_0 : \mathsf{Tm}\,\Gamma\,\mathsf{Ty}_0 \to \mathsf{Ty}\,\Gamma$.

- Ty_0 may be closed under arbitrary type formers, however, it is only possible to eliminate from Ty_0 type formers to types in Ty_0.

Types in Ty_0 are called *inner* types, while other types are *outer*. Alternatively, we may talk about *object-level* and *meta-level* types.

For example, if we have inner functions, we have the following:

$$\Pi_0 \qquad\quad : (A : \mathsf{Tm}\,\Gamma\,\mathsf{Ty}_0) \to \mathsf{Tm}\,(\Gamma \triangleright \mathsf{Tm}_0\,A) \to \mathsf{Tm}\,\Gamma\,\mathsf{Ty}_0$$

$$(\mathsf{app}_0, \mathsf{lam}_0) : \mathsf{Tm}\,\Gamma\,(\mathsf{Tm}_0\,(\Pi_0\,A\,B)) \simeq \mathsf{Tm}\,(\Gamma \triangleright \mathsf{Tm}_0\,A)\,(\mathsf{Tm}_0\,B)$$

If we have inner Booleans, we have the following (with β-rules omitted):

$$
\begin{aligned}
\mathsf{Bool}_0 \quad &: \mathsf{Tm}\,\Gamma\,\mathsf{Ty}_0 \\
\mathsf{true}_0 \quad &: \mathsf{Tm}\,\Gamma\,(\mathsf{Tm}_0\,\mathsf{Bool}_0) \\
\mathsf{false}_0 \quad &: \mathsf{Tm}\,\Gamma\,(\mathsf{Tm}_0\,\mathsf{Bool}_0) \\
\mathsf{BoolInd}_0 &: (B : \mathsf{Tm}\,(\Gamma \triangleright b : \mathsf{Tm}_0\,\mathsf{Bool}_0)\,\mathsf{Ty}_0) \\
&\quad \to \mathsf{Tm}\,\Gamma\,(\mathsf{Tm}_0\,(B[b \mapsto \mathsf{true}_0])) \\
&\quad \to \mathsf{Tm}\,\Gamma\,(\mathsf{Tm}_0\,(B[b \mapsto \mathsf{false}_0])) \\
&\quad \to (t : \mathsf{Tm}\,\Gamma\,(\mathsf{Tm}_0\,\mathsf{Bool}_0)) \to \mathsf{Tm}\,\Gamma\,(\mathsf{Tm}_0\,(B[b \mapsto t]))
\end{aligned}
$$

Intuitively, we can view outer types and terms as metatheoretical, while Ty_0 represents the set of types in the object theory, and Tm_0 witnesses that any object type can be mapped to a metatheoretical set of object terms. The restriction on elimination is crucial. If we have a Boolean term in the object language, we can use the object-level elimination principle to construct new object terms. But it makes no sense to eliminate into the metatheory. In fact, an object-level Boolean term is not necessarily true or false, it can also be just a variable or neutral term in some context, or it can be an arbitrary non-canonical value in a given model.

We review some properties of 2LTT. An important point is the action of Tm_0 on type formers. In general, Tm_0 preserves the negative type formers but not others.

For example, we have the isomorphism $\mathsf{Tm}_0\,(\Pi_0\,A\,B) \simeq \Pi_1\,(\mathsf{Tm}_0\,A)\,(\mathsf{Tm}_0\,B)$, where Π_1 denotes outer functions. We move left-to-right by mapping t to $\mathsf{lam}_1\,(\mathsf{app}_1\,t)$, and the other way by mapping t to $\mathsf{lam}_0\,(\mathsf{app}_0\,t)$. The preservation of Σ, $\top$, K and extensional identity is analogous.

In contrast, we can map from outer positive types to inner ones, but not the other way around. From $b : \mathsf{Tm}\,\Gamma\,\mathsf{Bool}_1$, we can use the outer Bool_1 recursor to return in $\mathsf{Tm}_0\,\mathsf{Bool}_0$. In the other direction, we only have constant functions since the Bool_0 recursor only targets types in Ty_0.

It may be the case that there are universes in the inner layer. For example, disregarding size issues (or just accepting an inconsistent inner theory), there may be an U_0 in Ty_0 such that we have $\mathsf{Tm}\,\Gamma\,(\mathsf{Tm}_0\,\mathsf{U}_0) = \mathsf{Tm}\,\Gamma\,\mathsf{Ty}_0$. This amounts to having a Russell-style inner universe with type-in-type. Assume that we have U_1 as well, as a meta-level Russell universe. Then we can map from $\mathsf{Tm}_0\,\mathsf{U}_0$ to U_1, by taking A to $\mathsf{Tm}_0\,A$, but we cannot map in the other direction.

3.3.2   Internal Syntax and Notation

In the rest of this thesis we will mostly work internally to a 2LTT, i.e. we use 2LTT as metatheory. We adapt the metatheoretical notations used up until now. We list used features and conventions below.

- We keep previous notation for type formers. For instance, Π-types are written as $(x : A) \to B$ or as $A \to B$.

- We assume a Coquand-style universe in the outer layer, named Set. As before, we leave the sizing levels implicit; if we were fully precise, we would write Set_i for a hierarchy of outer universes. Despite having a Coquand universe, we shall omit encoding and decoding in the internal syntax, and instead work in Russell-style. In practical implementations, elaborating Russell-style notation to Coquand-style is straightforward to do.

- If the same type formers are supported both in the inner and outer layers, we may distinguish them by $_0$ and $_1$ subscripts, e.g. by having Bool_0 and Bool_1. We omit some inferable subscripts, e.g. for Π and Σ-types. In these cases, we usually know from the type parameters which type former is meant. For example, $\mathsf{Tm}_0 \, \mathsf{Bool}_0 \to \mathsf{Bool}_1$ can only refer to outer functions.

- We have the convention that $- = -$ refers to the inner equality type, while $- \equiv -$ refers to the outer equality type. In the semantics of 2LTT that we use, outer equality of inner values is interpreted as definitional equality of inner terms, hence the naming scheme.

- By having Set, we are able to have $\mathsf{Ty}_0 : \mathsf{Set}$ and $\mathsf{Tm}_0 : \mathsf{Ty}_0 \to \mathsf{Set}$. So we don't have to deal with proper meta-level types, and have a more uniform notation. Notation and specification for inner type formers changes accordingly. For example, for inner Π-types we may write $(x : A) \to B$ if $A : \mathsf{Ty}_0$ and B depends on $x : \mathsf{Tm}_0 \, A$. This also enables a higher-order specification: if $B : \mathsf{Tm}_0 \, A \to \mathsf{Ty}_0$, then $(x : A) \to B \, x : \mathsf{Ty}_0$, and the specifying isomorphism for Π can be written as $\mathsf{Tm}_0 \, ((x : A) \to B \, x) \simeq ((x : \mathsf{Tm}_0 \, A) \to \mathsf{Tm}_0 \, (B \, x))$.

  Notation 8. An explicit notation for inner function abstraction would look like $\mathsf{lam}_0 \, t$ for $t : (x : \mathsf{Tm}_0 \, A) \to \mathsf{Tm}_0 \, (B \, x)$. This results in "double" abstraction, e.g. in $\mathsf{lam}_0 \, (\lambda \, x. \, \mathsf{suc}_0 \, (\mathsf{suc}_0 \, x)) : \mathsf{Tm}_0 \, (\mathsf{Nat}_0 \to \mathsf{Nat}_0)$. Instead of

this, we write $\lambda_0 \, x. \, t$ as a notation, thus we write $\lambda_0 \, x. \, \mathsf{suc}_0 \, (\mathsf{suc}_0 \, x)$ for the above example. We may also group multiple λ_0 binders together the same way as with λ.

- We may omit inferable Tm_0 applications. For instance, $\mathsf{Bool}_1 \to \mathsf{Bool}_0$ can be "elaborated" to $\mathsf{Bool}_1 \to \mathsf{Tm}_0 \, \mathsf{Bool}_0$ without ambiguity, since the function codomain must be on the same level as the domain, and the only thing we can do to make sense of this is to lift the codomain by Tm_0. Sometimes there is some ambiguity: $(\mathsf{Bool}_0 \to \mathsf{Bool}_0) \to \mathsf{Bool}_1$ can be elaborated both to $\mathsf{Tm}_0 \, (\mathsf{Bool}_0 \to \mathsf{Bool}_0) \to \mathsf{Bool}_1$ and to $(\mathsf{Tm}_0 \, \mathsf{Bool}_0 \to \mathsf{Tm}_0 \, \mathsf{Bool}_0) \to \mathsf{Bool}_1$. However, in this case the two output types are definitionally isomorphic, because of the Π-preservation by Tm_0. Hence, the elaboration choice does not make much difference, so we may still omit Tm_0-s in situations like this.

Example 7. Working in the internal syntax of 2LTT, the specification of Bool_0 looks like the following (omitting β again):

$$
\begin{array}{ll}
\mathsf{Bool}_0 & : \mathsf{Ty}_0 \\
\mathsf{true}_0 & : \mathsf{Bool}_0 \\
\mathsf{false}_0 & : \mathsf{Bool}_0 \\
\mathsf{BoolInd}_0 & : (B : \mathsf{Bool}_0 \to \mathsf{Ty}_0) \to B \, \mathsf{true}_0 \to B \, \mathsf{false}_0 \to (t : \mathsf{Bool}_0) \to B \, t
\end{array}
$$

If we elaborate the type of $\mathsf{BoolInd}_0$, we get the following:

$$
\begin{aligned}
\mathsf{BoolInd}_0 : {} & (B : \mathsf{Tm}_0 \, \mathsf{Bool}_0 \to \mathsf{Ty}_0) \to \mathsf{Tm}_0 \, (B \, \mathsf{true}_0) \to \mathsf{Tm}_0 \, (B \, \mathsf{false}_0) \\
& \to (t : \mathsf{Tm}_0 \, \mathsf{Bool}_0) \to \mathsf{Tm}_0 \, (B \, t)
\end{aligned}
$$

Here, the type is forced to live in the outer level because of the dependency on Ty_0. Since Ty_0 is an outer type, $\mathsf{Bool}_0 \to \mathsf{Ty}_0$ must be lifted, which in turn requires all other types to be lifted as well.

3.3.3 Alternative Presentation for 2LTT

We digress a bit on a different way to present 2LTT. In the primary 2LTT reference [ACKS19], inner and outer layers are specified as follows. We have two different *family structures* on the base cwf, $(\mathsf{Ty}_0, \mathsf{Tm}_0)$ and $(\mathsf{Ty}_1, \mathsf{Tm}_1)$, and a morphism between them. A family morphism is natural transformation mapping

types to types and terms to terms, which is an isomorphism on terms. We might
name the component maps as follows:

$$\Uparrow : \mathsf{Ty}_0\,\Gamma \to \mathsf{Ty}_1\,\Gamma$$

$$\uparrow : \mathsf{Tm}_0\,\Gamma\,A \to \mathsf{Tm}_1\,\Gamma\,(\Uparrow A)$$

$$\downarrow : \mathsf{Tm}_1\,\Gamma\,(\Uparrow A) \to \mathsf{Tm}_0\,\Gamma\,A$$

An advantage of this presentation is that we may close $(\mathsf{Ty}_0, \mathsf{Tm}_0)$ under type
formers without any encoding overhead, for example by having $\mathsf{Bool}_0 : \mathsf{Ty}_0\,\Gamma$,
$\mathsf{true}_0 : \mathsf{Tm}_0\,\Gamma\,\mathsf{Bool}_0$, etc., without the Tarski-style decoding. On the other hand,
we don't automatically get an outer universe of inner types. We can recover that
in two ways:

- We can assume an inner universe $\mathsf{U}_0 : \mathsf{Ty}_0\,\Gamma$, which can be lifted to the
  outer theory as $\Uparrow \mathsf{U}_0$. However, we may not want to make this assumption,
  in order to keep the inner theory as simple as possible.

- We can assume an outer universe which classifies elements of $\mathsf{Ty}_0\,\Gamma$. This
  amounts to reproducing the Ty_0 *type* from our 2LTT presentation, as an
  additional assumption. But in this case, we might as well skip the two
  family structures and the $\Uparrow$ morphism.

In this thesis we make ubiquitous use of the outer universe of inner types, so we
choose that to be the primitive notion, instead of having two family structures.

Do we lose anything by this? For the purposes of this thesis, not really.
However, if we want to implement 2LTT as a system for two-stage compilation,
the $\Uparrow$ syntax appears to be closer to existing systems. Staging is about com-
puting all outer redexes but no inner ones, thereby outputting syntax which is
purely in the inner theory. This can be implemented as a stage-aware variant of
normalization-by-evaluation [?]. We can give an intuitive staging interpretation
for the operators in the $\Uparrow$ syntax:

- $\Uparrow A$ is the type of A-expressions. This corresponds to $a\ \mathsf{code}$ in MetaOcaml
  [?] and $\mathsf{Exp}\,a$ in Template Haskell.

- $\uparrow$ is *quoting*, which creates an expression from any inner term. This is $.\langle - \rangle.$
  in MetaOCaml and $[|-|]$ in Template Haskell.

- $\downarrow$ is *splicing*, which inserts the result of a meta-level computation into an
  object-level expression. This is $\sim$ in MetaOCaml and $ in Template Haskell.

For example, in the $\Uparrow$ syntax, we might write a polymorphic identity function which acts on inner types in two different ways:

$$\mathsf{id} : (A : \mathsf{U}_0) \to A \to A \qquad \mathsf{id}' : (A : \Uparrow \mathsf{U}_0) \to \Uparrow (\downarrow A) \to \Uparrow (\downarrow A)$$

$$\mathsf{id} :\equiv \lambda_0 \, A \, x. \, x \qquad \mathsf{id}' :\equiv \lambda_1 \, A \, x. \, x$$

The first one lives in the inner family structure. The second one is the same thing, but lifted to the outer theory. The choice between the two allows us to control staging-time evaluation. If we write $\mathsf{id} \, \mathsf{Bool}_0 \, \mathsf{true}_0$, that's an inner expression which goes into the staging output as it is. On the other hand, $\downarrow (\mathsf{id}' \, (\uparrow \mathsf{Bool}_0) \, (\uparrow \mathsf{true}_0))$ reduces to $\downarrow (\uparrow \mathsf{true}_0)$ which in turn reduces to true_0. The same choice can be expressed in our syntax as well:

$$\mathsf{id} : \mathsf{Tm}_0 \, ((A : \mathsf{U}_0) \to A \to A) \qquad \mathsf{id}' : (A : \mathsf{Tm}_0 \, \mathsf{U}_0) \to \mathsf{Tm}_0 \, A \to \mathsf{Tm}_0 \, A$$

$$\mathsf{id} :\equiv \lambda_0 \, A \, x. \, x \qquad \mathsf{id}' :\equiv \lambda \, A \, x. \, x$$

It remains to be checked which style is preferable in a staging implementation. In the $\Uparrow$ style, the quoting and splicing operations add noise to core syntax, but they are also mostly inferable during elaboration, and they pack stage-changing information into $\uparrow$ and $\downarrow$, thereby making it feasible to omit stage annotations in other places in the core syntax. In the Ty_0 style, we don't have quote/splice, but we have to keep track of stages in all type/term formers. The author of this thesis has made a prototype implementation of staging in the $\Uparrow$ style but not in the Ty_0 style [?]. It would be interesting to compare the two in future work.

3.4   Standard Semantics of 2LTT

We review the standard semantics of 2LTT which we use in the rest of the thesis. This justifies the metaprogramming view, that 2LTT allows meta-level reasoning about an inner theory.

We present it two steps, by assuming progressively more structure in the inner theory. First, we only assume a category. This already lets us present a presheaf semantics for the outer layer. Then, we assume a cwf as the inner theory, which lets us interpret Ty_0 and Tm_0 and also consider inner type formers.

3.4.1 Presheaf Model of the Outer Layer

In this subsection we present a presheaf model for the outer layer of 2LTT, that is, the base category together with the terminal object, the $(\mathsf{Ty}, \mathsf{Tm})$ family and some type formers. This presheaf semantics is well-known in the literature [?]. We give a specification which follows [?] and [?] most closely.

In the following, we work outside 2LTT (since we are defining a model of 2LTT), in a suitable metatheory; an extensional type theory with enough Set universes suffices.

We assume a $\mathbb{C}$ category. We write i, j, $k : |\mathbb{C}|$ for objects and f, g, $h : \mathbb{C}(i, j)$ for morphisms. We use a different notation than for cwfs before, in order to disambiguate components in $\mathbb{C}$ from components in the presheaf model of 2LTT. We use $\hat{\mathbb{C}}$ to refer to the model which is being defined. We use the same component names for $\hat{\mathbb{C}}$ as in Section 3.2.

Model of Cwf

Definition 23. $\Gamma : \mathsf{Con}$ is a presheaf over $\mathbb{C}$. Its components are as follows.

$$
\begin{aligned}
&|\Gamma| &&: |\mathbb{C}| \to \mathsf{Set} \\
&-\langle - \rangle &&: |\Gamma|\, j \to \mathbb{C}(i, j) \to |\Gamma|\, i \\
&\gamma\langle \mathsf{id} \rangle &&= \gamma \\
&\gamma\langle f \circ g \rangle = \gamma\langle f \rangle\langle g \rangle
\end{aligned}
$$

We flip around the order of arguments in the action of Γ on morphisms. This is more convenient because of the contravariance; we can observe this in the statement of preservation laws already. The action on morphisms is sometimes called *restriction*.

Definition 24. $\sigma : \mathsf{Sub}\,\Gamma\,\Delta$ is a natural transformation from Γ to Δ. It has action $|\sigma| : |\Gamma|\, i \to |\Delta|\, i$, such that $|\sigma|(\gamma\langle f \rangle) = (|\sigma|\gamma)$.

Definition 25. $A : \mathsf{Ty}\,\Gamma$ is a displayed presheaf over Γ. The "displayed" here is used in exactly the same sense as in "displayed algebra" before. As we will see in Chapter 4, presheaves can be specified with a signature, in which case a presheaf is an algebra, and a displayed presheaf is a displayed algebra. The definition here is equivalent to saying that A is a presheaf over the category of elements of Γ,

but it is more convenient to use in concrete definitions and calculations. The components of A are as follows.

$$
\begin{aligned}
|A| \quad & : |\Gamma|\, i \to \mathsf{Set} \\
-\langle - \rangle \quad & : |A|\, \gamma \to (f : \mathbb{C}(i,\, j)) \to |A|\, (\gamma\langle f \rangle) \\
\alpha\langle \mathsf{id} \rangle \quad & = \alpha \\
\alpha\langle f \circ g \rangle & = \alpha\langle f \rangle\langle g \rangle
\end{aligned}
$$

Definition 26. $t : \mathsf{Tm}\,\Gamma\,A$ is a section of the displayed presheaf A. This is again the same notion of section that we have seen before, instantiated for presheaves.

$$
\begin{aligned}
|t| &: (\gamma : |\Gamma|)\, i \to |A|\, \gamma \\
|t|(\gamma\langle f \rangle) &= (|t|\gamma)\langle f \rangle
\end{aligned}
$$

Definition 27. $\Gamma \rhd A : \mathsf{Con}$ is the total presheaf of the displayed presheaf A. Its action on objects and morphisms is the following.

$$
\begin{aligned}
|\Gamma \rhd A| \quad & :\equiv (\gamma : |\Gamma|) \times |A\,\gamma| \\
(\gamma,\, \alpha)\langle f \rangle & :\equiv (\gamma\langle f \rangle,\, \alpha\langle f \rangle)
\end{aligned}
$$

The id and $-\circ-$ preservation laws follow immediately.

Definition 28. $A[\sigma] : \mathsf{Ty}\,\Gamma$ is defined as follows, assuming $A : \mathsf{Ty}\,\Delta$ and $\sigma : \mathsf{Sub}\,\Gamma\,\Delta$.

$$
\begin{aligned}
|A[\sigma]|\, \gamma & :\equiv |A|\, (|\sigma|\, \gamma) \\
\alpha\langle f \rangle \quad & :\equiv \alpha\langle f \rangle
\end{aligned}
$$

In the second component, we use $-\langle - \rangle$ for A on the right hand side. The definition is well-typed since $|A|\, (|\sigma|\, (\gamma\langle f \rangle)) = |A|\, ((|\sigma|\, \gamma)\langle f \rangle)$ by the naturality of σ. Functoriality follows from functoriality of A.

It's easy to check that the above definitions can be extended to a cwf.

- For the base category, we take the category of presheaves.

- The empty context $\bullet$ is the terminal presheaf, i.e. the constantly $\top$ functor.

- Type substitution is functorial, as it is defined as simple function composition of actions on objects.

- Term substitution is defined as composition of a section and a natural transformation; and also functorial for the same reason.

- Context comprehension structure follows from the Σ-based definition for context extension.

Yoneda Embedding

Before continuing with interpreting type formers in $\hat{\mathbb{C}}$, we review the Yoneda embedding, as it is useful in subsequent definitions.

Definition 29. The **Yoneda embedding**, denoted y, is a functor from $\mathbb{C}$ to the underlying category of $\hat{\mathbb{C}}$, defined as follows.

$$\mathsf{y} : |\mathbb{C}| \to \mathsf{Con} \qquad \mathsf{y} : \mathbb{C}(i,\, j) \to \mathsf{Sub}\,(\mathsf{y}\,i)\,(\mathsf{y}\,j)$$

$$\mathsf{y}\,i :\equiv \mathbb{C}(-,\, i) \qquad |\mathsf{y}\,f|\,g :\equiv f \circ g$$

Lemma 1 (Yoneda lemma). We have $\mathsf{Sub}\,(\mathsf{y}\,i)\,\Gamma \simeq |\Gamma|\,i$ as an isomorphism of sets, natural in i [**?**].

Corollary. If we choose Γ to be $\mathsf{y}j$, it follows that $\mathsf{Sub}\,(\mathsf{y}\,i)\,(\mathsf{y}\,j) \simeq \mathbb{C}(i,\, j)$, i.e. that y is bijective on morphisms; hence it is an embedding.

Notation 9. For $\gamma : |\Gamma|\,i$, we use $\gamma\langle - \rangle : \mathsf{Sub}\,(\mathsf{y}\,i)\,\Gamma$ to denote transporting right-to-left along the Yoneda lemma. In the other direction we don't really need a notation, since from $\sigma : \mathsf{Sub}\,(\mathsf{y}\,i)\,\Gamma$ we get $\sigma\,\mathsf{id} : |\Gamma|\,i$.

Type Formers

Definition 30. Constant families are displayed presheaves which do not depend on their context.

$$\mathsf{K} \qquad\qquad : \mathsf{Con} \to \{\Gamma : \mathsf{Con}\} \to \mathsf{Ty}\,\Gamma$$

$$|\mathsf{K}\,\Delta|\,\{i\}\,\gamma :\equiv |\Delta|\,i$$

$$\delta\langle f \rangle \qquad\quad :\equiv \delta\langle f \rangle$$

With this definition, we have $\mathsf{Tm}\,\Gamma\,(\mathsf{K}\,\Delta) = \mathsf{Sub}\,\Gamma\,\Delta$ so we have strict constant families.

Notation 10. It is useful to consider any set as a constant presheaf, so given $A : \mathsf{Set}$ we may write $A : \mathsf{Con}$ for the constant presheaf as well.

Definition 31. From any $A : \mathsf{Set}$, we get $\mathsf{K}\,A : \mathsf{Ty}\,\Gamma$. This can be used to model negative or positive **closed type formers**. For example, natural numbers are modeled as $\mathsf{K}\,\mathbb{N}$, Booleans as $\mathsf{K}\,\mathsf{Bool}$, the unit type as $\mathsf{K}\,\top$, and so on.

Definition 32. Coquand universes can be defined as follows. We write $\mathsf{Set}_{\hat{\mathbb{C}}}$ for the outer universe in the model, to distinguish it from the external Set. Since the $\mathsf{Set}_{\hat{\mathbb{C}}}$ is a non-dependent type, it is helpful to define it as a $\mathsf{Set}_{\hat{\mathbb{C}}} : \mathsf{Con}$ such that $\mathsf{Sub}\,\Gamma\,\mathsf{Set}_{\hat{\mathbb{C}}} \simeq \mathsf{Ty}\,\Gamma$. The usual universe can be derived from this as $\mathsf{K}\,\mathsf{Set}_{\hat{\mathbb{C}}}$. Again, we ignore size issues; the fully formal definition would involve indexing constructions in $\hat{\mathbb{C}}$ by universe levels.

We can take a hint from the Yoneda lemma. We aim to define $|\mathsf{Set}_{\hat{\mathbb{C}}}|\,i$, but by the Yoneda lemma it is isomorphic to $\mathsf{Sub}\,(\mathsf{y}i)\,\mathsf{Set}_{\hat{\mathbb{C}}}$. However, by specification this should be isomorphic to $\mathsf{Ty}\,(\mathsf{y}\,i)$, so we take this as definition:

$$\mathsf{Set}_{\hat{\mathbb{C}}} \quad : \mathsf{Con}$$
$$|\mathsf{Set}_{\hat{\mathbb{C}}}|\,i :\equiv \mathsf{Ty}\,(\mathsf{y}\,i)$$
$$A\langle f\rangle \quad :\equiv A[\mathsf{y}f]$$

In the $A\langle f\rangle$ definition, we substitute $A : \mathsf{Ty}\,(\mathsf{y}\,i)$ with $\mathsf{y}f : \mathsf{Sub}\,(\mathsf{y}j)\,(\mathsf{y}i)$ to get an element of $\mathsf{Ty}\,(\mathsf{y}j)$. The required $\mathsf{Sub}\,\Gamma\,\mathsf{Set}_{\hat{\mathbb{C}}} \simeq \mathsf{Ty}\,\Gamma$ is straightforward, so we omit the definition.

We note that *Russell* universes are not supported in the outer layer, as $\mathsf{Sub}\,\Gamma\,\mathsf{Set}_{\hat{\mathbb{C}}}$ and $\mathsf{Ty}\,\Gamma$ are not strictly the same, in particular they have a different number of components as iterated Σ-types. Nevertheless, as we mentioned in Section 3.3.2, we use Russell-style notation in the internal 2LTT syntax, and assume that encoding/decoding is inserted by elaboration.

Definition 33. Σ-types are defined pointwise. The definitions for pairing and projections follow straightforwardly.

$$\Sigma \quad\quad : (A : \mathsf{Ty}\,\Gamma) \to \mathsf{Ty}\,(\Gamma \triangleright A) \to \mathsf{Ty}\,\Gamma$$
$$|\Sigma\,A\,B|\,\gamma :\equiv (\alpha : |A|\,\gamma) \times |B|\,(\gamma,\,\alpha)$$
$$(\alpha,\,\beta)\langle f\rangle :\equiv (\alpha\langle f\rangle,\,\beta\langle f\rangle)$$

Definition 34. We define **Π-types** in the following. This is a bit more complicated, so first we look at the simpler case of presheaf exponentials.

The exponential $\Delta^{\Gamma} : \mathsf{Con}$ is characterized by the isomorphism $\mathsf{Sub}\,(\Gamma \otimes \Delta)\,\Xi \simeq \mathsf{Sub}\,\Gamma\,(\Xi^{\Delta})$, where we write $\otimes$ for the pointwise product of two presheaves. We

can again use the Yoneda lemma. We want to define $|\Delta^\Gamma|\, i$, but this is isomorphic to $\mathsf{Sub}\,(\mathsf{y}i)\,(\Delta^\Gamma)$, which should be isomorphic to $\mathsf{Sub}\,(\mathsf{y}i \otimes \Gamma)\,\Delta$ by the specification of exponentials. Hence:

$$|\Delta^\Gamma|\, i :\equiv \mathsf{Sub}\,(\mathsf{y}i \otimes \Gamma)\,\Delta$$

$$\sigma\langle f \rangle \quad :\equiv \sigma \circ (\mathsf{y}f \circ \mathsf{p}, \mathsf{q})$$

In the definition of presheaf restriction, we use p, q as projections and $-,-$ as pairing for $\otimes$. In short, $(\mathsf{y}f \circ \mathsf{p}, \mathsf{q})$ is the same as the morphism lifting from Definition 13: it weakens $\mathsf{y}f : \mathsf{Sub}\,(\mathsf{y}j)\,(\mathsf{y}i)$ to $\mathsf{Sub}\,(\mathsf{y}j \otimes \Gamma)\,(\mathsf{y}i \otimes \Gamma)$.

The dependently typed case follows the same pattern, except that we use Tm and $-\triangleright-$ instead of Sub and $-\otimes-$. Additionally, the action on objects depends on $\gamma : |\Gamma|\, i$, and we make use of $\gamma\langle-\rangle : \mathsf{Sub}\,(\mathsf{y}i)\,\Gamma$ (introduced in Notation 9).

$$\Pi \qquad\qquad : (A : \mathsf{Ty}\,\Gamma) \to \mathsf{Ty}\,(\Gamma \triangleright A) \to \mathsf{Ty}\,\Gamma$$

$$|\Pi\,A\,B|\,\{i\}\,\gamma :\equiv \mathsf{Tm}\,(\mathsf{y}i \triangleright A[\gamma\langle-\rangle])\,(B[\gamma\langle-\rangle \circ \mathsf{p}, \mathsf{q}])$$

$$t\langle f \rangle \qquad\quad :\equiv t[\mathsf{y}f \circ \mathsf{p}, \mathsf{q}]$$

Let's unfold the above definition a bit. Assuming $t : |\Pi\,A\,B|\,\{i\}\,\gamma$, we have

$$|t| : \{j : |\mathbb{C}|\} \to ((f, \alpha) : (f : \mathbb{C}(j, i)) \times |A|\,(\gamma\langle f\rangle)) \to |B|\,(\gamma\langle f\rangle, \alpha)$$

This is a bit clearer if we remove the Σ-type by currying.

$$|t| : \{j : |\mathbb{C}|\}(f : \mathbb{C}(j, i))(\alpha : |A|\,(\gamma\langle f\rangle)) \to |B|\,(\gamma\langle f\rangle, \alpha)$$

Restriction is functorial since it is defined as Tm substitution. The definitions for lam and app are left to the reader.

Definition 35. Extensional identity is defined as pointwise equality of sections:

$$\mathsf{Id} : \mathsf{Tm}\,\Gamma\,A \to \mathsf{Tm}\,\Gamma\,A \to \mathsf{Ty}\,\Gamma$$

$$|\mathsf{Id}\,t\,u|\,\gamma :\equiv |t|\,\gamma = |u|\,\gamma$$

For the restriction operation, we have to show that $|t|\,\gamma = |u|\,\gamma$ implies $|t|\,(\gamma\langle f\rangle) = |u|\,(\gamma\langle f\rangle)$. This follows from congruence by $-\langle f\rangle$ and naturality of t and u. The defining $(\mathsf{reflect}, \mathsf{refl}) : \mathsf{Tm}\,\Gamma\,(\mathsf{Id}\,t\,u) \simeq (t = u)$ isomorphism is evident, assuming UIP and function extensionality for the metatheoretic $-=-$ relation (which we do assume).

3.4.2 Modeling The Inner Layer

We assume now that $\mathbb{C}$ is a cwf. We write types as a, b, c : $\mathsf{Ty}_\mathbb{C}\, i$ and terms as t, u, v : $\mathsf{Tm}_\mathbb{C}\, i\, a$. We reuse $\bullet$ for the terminal object and $- \triangleright -$ for context extension, and likewise reuse notation for substitutions.

Definition 36 (Ty_0, Tm_0). First, note that $\mathsf{Ty}_\mathbb{C}$ is a presheaf over $\mathbb{C}$, and $\mathsf{Tm}_\mathbb{C}$ is a displayed presheaf over $\mathsf{Ty}_\mathbb{C}$; this follows from the requirement that they form a family structure over $\mathbb{C}$. Hence, in the presheaf model $\mathsf{Ty}_\mathbb{C}$ is an element of Con and $\mathsf{Tm}_\mathbb{C}$ is an element of $\mathsf{Ty}\,\mathsf{Ty}_\mathbb{C}$. Also recall from Definition 30 that $\mathsf{Tm}\,\Gamma\,(\mathsf{K}\,\Delta) = \mathsf{Sub}\,\Gamma\,\Delta$. With this is mind, we give the following definitions:

$$\mathsf{Ty}_0 : \mathsf{Ty}\,\Gamma \qquad \mathsf{Tm}_0 : \mathsf{Tm}\,\Gamma\,\mathsf{Ty}_0 \to \mathsf{Ty}\,\Gamma$$

$$\mathsf{Ty}_0 :\equiv \mathsf{K}\,\mathsf{Ty}_\mathbb{C} \qquad \mathsf{Tm}_0\,A :\equiv \mathsf{Tm}_\mathbb{C}[A]$$

$\mathsf{Tm}_\mathbb{C}[A]$ is well-typed since $A : \mathsf{Tm}\,\Gamma\,(\mathsf{K}\,\mathsf{Ty}_\mathbb{C})$, thus $A : \mathsf{Sub}\,\Gamma\,\mathsf{Ty}_\mathbb{C}$. In other words, A is a natural transformation from Γ to the presheaf of inner types.

Inner Type Formers

Can type formers in ($\mathsf{Ty}_\mathbb{C}$, $\mathsf{Tm}_\mathbb{C}$) be transferred to (Ty_0, Tm_0) in the presheaf model of 2LTT? For example, if $\mathbb{C}$ supports Bool, we would like to model Bool_0 in Ty_0 as well. The following explanation is adapted from Capriotti [Cap17, Section 2.3].

Generally, a type former in $\mathbb{C}$ transfers to $\hat{\mathbb{C}}$ if it can be specified in the internal language of $\hat{\mathbb{C}}$; if the type former "always has been" in $\hat{\mathbb{C}}$ to begin with. To be describable in $\hat{\mathbb{C}}$, a type former needs to be natural with respect to $\mathbb{C}$ morphisms. This is also a core idea of HOAS: when working in $\hat{\mathbb{C}}$, everything is natural, and we can omit boilerplate related to contexts and substitutions. For example, consider the specification of inner Π-types in the internal syntax of 2LTT:

$$\Pi_0 \qquad\qquad : (A : \mathsf{Ty}_0) \to (\mathsf{Tm}_0\,A \to \mathsf{Ty}_0) \to \mathsf{Ty}_0$$

$$(\mathsf{app}_0, \mathsf{lam}_0) : \Pi_0\,A\,B \simeq ((a : \mathsf{Tm}_0\,A) \to \mathsf{Tm}_0\,(B\,a))$$

We can say that this *defines* what it means for $\mathbb{C}$ to support Π. We recover the usual non-higher-order specification of Π in the following way, up to isomorphism:

- First, we interpret the higher-order specification as a context or closed Σ-type in the standard presheaf model of 2LTT. This yields a presheaf over $\mathbb{C}$.

- Then, we evaluate the resulting presheaf at the terminal object $\bullet$. This yields a set which is isomorphic to the conventional specification of Π.

In summary, if by "type formers" we mean extra structure on $(\mathsf{Ty}_0, \mathsf{Tm}_0)$ which is definable in 2LTT, then *by definition* all such type formers transfer from $\mathbb{C}$ to $(\mathsf{Ty}_0, \mathsf{Tm}_0)$. This holds for every type former mentioned in this thesis.

3.4.3   Functions With Inner Domains

There is a useful semantic simplification in the standard presheaf model, in cases where we have functions of the form $\Pi\,(\mathsf{Tm}_0\,A)\,B$. This greatly reduces encoding overhead when interpreting inductive signatures in 2LTT; we look at examples in Section 3.5. First we look at the simply-typed case with presheaf exponentials.

Lemma 2. y preserves finite products up to isomorphism, i.e. $\mathsf{y}\bullet \simeq \bullet$ and $\mathsf{y}(i\otimes j) \simeq (\mathsf{y}i \otimes \mathsf{y}j)$.

Proof. $\mathsf{y}\bullet$ is $\mathbb{C}(-, \bullet)$ by definition, which is pointwise isomorphic to $\top$, hence isomorphic to $\bullet \equiv \mathsf{K}\,\top$. $\mathsf{y}(i \otimes j)$ is $\mathbb{C}(-, i \otimes j)$, which is isomorphic to $\mathsf{y}i \otimes \mathsf{y}j$ by the specification of products. $\qquad\qquad\square$

Lemma 3. We have the following isomorphism.

$$|\Gamma^{\mathsf{y}i}|\,j \equiv$$
$$\mathsf{Sub}\,(\mathsf{y}j \otimes \mathsf{y}i)\,\Gamma \simeq \qquad \text{by product preservation}$$
$$\mathsf{Sub}\,(\mathsf{y}(j \otimes i))\,\Gamma \simeq \qquad \text{by Yoneda lemma}$$
$$|\Gamma|\,(j \otimes i)$$

It is possible to rephrase the above derivation for Π-types. For that, we would need to define the action of y on types and terms, consider the preservation of $-\rhd-$ by y, and also specify a "dependent" Yoneda lemma for Tm. For the sake of brevity, we omit this, and present the result directly:

$$|\Pi\,(\mathsf{Tm}_0\,A)\,B|\,\{i\}\,\gamma \simeq |B|\,\{i \rhd |A|\,\gamma\}\,(\gamma\langle\mathsf{p}\rangle, \mathsf{q})$$

In short, depending on an inner domain is the same as depending on an extended context in $\mathbb{C}$. We expand a bit on the typing of the right hand side. We have $\gamma : |\Gamma|\,i$, moreover

$$|B| \qquad\qquad : \{j : \mathbb{C}\} \to |\Gamma \rhd \mathsf{Tm}_0\,A|\,j \to \mathsf{Set}$$

$$|B| \qquad : \{j : \mathbb{C}\} \to ((\gamma' : |\Gamma|\, j) \times \mathsf{Tm}_{\mathbb{C}}\, j\, (|A|\, \gamma')) \to \mathsf{Set}$$

$$|B|\, \{i \triangleright |A|\, \gamma\} : ((\gamma' : |\Gamma|\, (i \triangleright |A|\, \gamma)) \times \mathsf{Tm}_{\mathbb{C}}\, (i \triangleright |A|\, \gamma)\, (|A|\, \gamma')) \to \mathsf{Set}$$

$$\gamma \langle \mathsf{p} \rangle \qquad : |\Gamma|\, (i \triangleright |A|\, \gamma)$$

$$\mathsf{q} \qquad : \mathsf{Tm}_{\mathbb{C}}\, (i \triangleright |A|\, \gamma)\, ((|A|\, \gamma)[\mathsf{p}])$$

$$\mathsf{q} \qquad : \mathsf{Tm}_{\mathbb{C}}\, (i \triangleright |A|\, \gamma)\, (|A|\, (\gamma \langle \mathsf{p} \rangle))$$

3.5 Simple Inductive Signatures in 2LTT

We revisit simple inductive signatures in this section, working internally to 2LTT. We review the concepts introduced in Chapter 2 in the same order.

Notation 11. In this section we shall be fairly explicit about writing Tm_0-s and transporting along definitional isomorphisms. The simple setting makes it feasible to be explicit; in later chapters we are more terse, as signatures and semantics get more complicated.

3.5.1 Theory of Signatures

Signatures are defined exactly in the same way as before: we have $\mathsf{Con} : \mathsf{Set}$, $\mathsf{Ty} : \mathsf{Set}$, $\mathsf{Sub} : \mathsf{Con} \to \mathsf{Con} \to \mathsf{Set}$, $\mathsf{Var} : \mathsf{Con} \to \mathsf{Ty} \to \mathsf{Set}$ and $\mathsf{Tm} : \mathsf{Con} \to \mathsf{Ty} \to \mathsf{Set}$. However, now by Set we mean the outer universe of 2LTT. Thus signatures are inductively defined in the outer layer.

3.5.2 Algebras

Again we compute algebras by induction on signatures, but now we use inner types for carriers of algebras. We interpret types as follows:

$$-^A : \mathsf{Ty} \to \mathsf{Ty}_0 \to \mathsf{Set}$$

$$\iota^A \qquad X :\equiv \mathsf{Tm}_0\, X$$

$$(\iota \to A)^A\, X :\equiv \mathsf{Tm}_0\, X \to A^A\, X$$

Elsewhere, we change the type of the X parameters accordingly:

$$-^A : \mathsf{Con} \to \mathsf{Ty}_0 \to \mathsf{Set}$$

$$-^A : \mathsf{Var}\,\Gamma\,A \to \{X : \mathsf{Ty}_0\} \to \Gamma^A\,X \to A^A\,X$$

$$-^A : \mathsf{Tm}\,\Gamma\,A \to \{X : \mathsf{Ty}_0\} \to \Gamma^A\,X \to A^A\,X$$

$$-^A : \mathsf{Sub}\,\Gamma\,\Delta \to \{X : \mathsf{Ty}_0\} \to \Gamma^A\,X \to \Delta^A\,X$$

We also define $\mathsf{Alg}\,\Gamma$ as $(X : \mathsf{Ty}_0) \times \Gamma^A\,X$.

Example 8. Inside 2LTT we have the following:[3]

$$\mathsf{Alg}\,\mathsf{NatSig} \equiv (X : \mathsf{Ty}_0) \times (\mathit{zero} : \mathsf{Tm}_0\,X) \times (\mathit{suc} : \mathsf{Tm}_0\,X \to \mathsf{Tm}_0\,X)$$

Then, we may assume any cwf $\mathbb{C}$, and interpret the above closed type in the presheaf model $\hat{\mathbb{C}}$, and evaluate the result at $\bullet$ and the unique element of the terminal presheaf $\mathsf{K}\,\top$:

$$|\mathsf{Alg}\,\mathsf{NatSig}|\,\{\bullet\}\,\mathsf{tt} : \mathsf{Set}$$

We only need to compute definitions now. We use the simplified semantics for $\mathit{suc} : \mathsf{Tm}_0\,X \to \mathsf{Tm}_0\,X$, since the function domain is an inner type. We get the following:

$$|\mathsf{Alg}\,\mathsf{NatSig}|\,\{\bullet\}\,\mathsf{tt} \equiv (X : \mathsf{Ty}_{\mathbb{C}}\,\bullet) \times (\mathit{zero} : \mathsf{Tm}_{\mathbb{C}}\,\bullet\,X) \times (\mathit{suc} : \mathsf{Tm}_{\mathbb{C}}\,(\bullet \triangleright X)\,X)$$

Using the same computation, we get the following for binary trees:

$$|\mathsf{Alg}\,\mathsf{TreeSig}|\,\{\bullet\}\,\mathsf{tt} \equiv (X : \mathsf{Ty}_{\mathbb{C}}\,\bullet) \times (\mathit{leaf} : \mathsf{Tm}_{\mathbb{C}}\,\bullet\,X) \times (\mathit{node} : \mathsf{Tm}_{\mathbb{C}}\,(\bullet \triangleright X \triangleright X)\,X)$$

We can also get internal algebras in any $\mathbb{C}$ category with finite products, because we can build cwfs from all such $\mathbb{C}$.

Definition 37. Assuming $\mathbb{C}$ with finite products, we build a cwf by setting $\mathsf{Con} :\equiv |\mathbb{C}|$, $\mathsf{Ty}\,\Gamma :\equiv |\mathbb{C}|$, $\mathsf{Sub}\,\Gamma\,\Delta :\equiv \mathbb{C}(\Gamma,\,\Delta)$, $\mathsf{Tm}\,\Gamma\,A :\equiv \mathbb{C}(\Gamma,\,A)$, $\Gamma \triangleright A :\equiv \Gamma \otimes A$ and $\bullet :\equiv \bullet_{\mathbb{C}}$. In short, we build a non-dependent (simply-typed) cwf.

Now we can effectively interpret signatures in finite product categories. For example:

$$|\mathsf{Alg}\,\mathsf{NatSig}|\,\{\bullet\}\,\mathsf{tt} \equiv (X : |\mathbb{C}|) \times (\mathit{zero} : \mathbb{C}(\bullet,\,X)) \times (\mathit{suc} : \mathbb{C}(\bullet \otimes X,\,X))$$

This is almost the same as what we would write by hand for the specification of natural number objects; the only difference is the extra $\bullet \otimes -$ in suc.

---

[3]Up to isomorphism, since we previously defined Γ^A as a function type instead of an iterated product type.

3.5.3 Morphisms

We get an additional degree of freedom in the computation of morphisms: preservation equations can be inner or outer. The former option is *weak* or *propositional* preservation, while the latter is *strict* preservation. In the presheaf model of 2LTT, outer equality is definitional equality of inner terms, while inner equality is propositional equality in the inner theory. Of course, if the inner theory has extensional identity types, weak and strict equations in 2LTT coincide for inner types. We compute weak preservation for types as follows.

$$-^M : (A : \mathsf{Ty})\{X_0\, X_1 : \mathsf{Ty}_0\}(X^M : \mathsf{Tm}_0\, X_0 \to \mathsf{Tm}_0\, X_1) \to A^A\, X_0 \to A^A\, X_1 \to \mathsf{Set}$$
$$\iota^M \qquad\quad X^M\, \alpha_0\, \alpha_1 :\equiv \mathsf{Tm}_0\, (X^M\, \alpha_0 = \alpha_1)$$
$$(\iota \to A)^M\, X^M\, \alpha_0\, \alpha_1 :\equiv (x : \mathsf{Tm}_0\, X_0) \to A^M\, X^M\, (\alpha_0\, x)\, (\alpha_1\, (X^M\, x))$$

For strict preservation, we simply change $\mathsf{Tm}_0\, (X^M\, \alpha_0 = \alpha_1)$ to $X^M\, \alpha_0 \equiv \alpha_1$. The definition of morphisms is the same as before:

$$-^M : (\Gamma : \mathsf{Con}_1)\{X_0\, X_1 : \mathsf{Ty}_0\} \to (\mathsf{Tm}_0\, X_0 \to \mathsf{Tm}_0\, X_1) \to \Gamma^A\, X_0 \to \Gamma^A\, X_1 \to \mathsf{Set}$$
$$\Gamma^M\, X^M\, \gamma_0\, \gamma_1 :\equiv \{A\}(x : \mathsf{Var}_1\, \Gamma\, A) \to A^M\, X^M\, (\gamma_0\, x)\, (\gamma_1\, x)$$

$$\mathsf{Mor} : \{\Gamma : \mathsf{Con}_1\} \to \mathsf{Alg}\, \Gamma \to \mathsf{Alg}\, \Gamma \to \mathsf{Set}$$
$$\mathsf{Mor}\, \{\Gamma\}\, (X_0,\, \gamma_0)\, (X_1,\, \gamma_1) :\equiv (X^M : \mathsf{Tm}_0\, X_0 \to \mathsf{Tm}_0\, X_1) \times \Gamma^M\, X^M\, \gamma_0\, \gamma_1$$

We omit here the $-^M$ definitions for terms and substitutions.

3.5.4 Displayed Algebras

We present $-^D$ only for types below.

$$-^D : (A : \mathsf{Ty})\{X\} \to (\mathsf{Tm}_0\, X \to \mathsf{Ty}_0) \to A^A\, X \to \mathsf{Set}$$
$$\iota^D \qquad\quad X^D\, \alpha :\equiv \mathsf{Tm}_0\, (X^D\, \alpha)$$
$$(\iota \to A)^D\, X^D\, \alpha :\equiv (x : \mathsf{Tm}_0\, X)(x^D : \mathsf{Tm}_0\, (X^D\, x)) \to A^D\, X^D\, (\alpha\, x)$$

We use type dependency in the inner layer. In the presheaf model, this is interpreted as inner types depending on certain contexts.

Example 9. Assume a closed $(X,\, zero,\, suc)$ Nat-algebra in 2LTT. We have the

following computation:

$$\mathsf{DispAlg}\,\{\mathsf{NatSig}\}\,(X,\ zero,\ suc) \equiv$$
$$(X^D \quad : \mathsf{Tm}_0\,X \to \mathsf{Ty}_0)$$
$$\times\,(zero^D : \mathsf{Tm}_0\,(X^D\,zero))$$
$$\times\,(suc^D \quad : (n : \mathsf{Tm}_0\,X) \to \mathsf{Tm}_0\,(X^D\,n) \to \mathsf{Tm}_0\,(X^D\,(suc\,n)))$$

Let's look at the presheaf interpretation now. We simplify functions with inner domains everywhere. Also note that for $\mathsf{suc} : \mathsf{Tm}_0\,X \to \mathsf{Tm}_0\,X$, we get $|\mathsf{suc}|\,\mathsf{tt} : \mathsf{Tm}_{\mathbb{C}}\,(\bullet \triangleright n : |X|\,\mathsf{tt})\,(|X|\,\mathsf{tt})$ in the semantics, so a $\mathsf{suc}\,t$ application is translated as a substitution $(|\mathsf{suc}|\,\mathsf{tt})[n \mapsto |t|\,\mathsf{tt}]$.

$$|\mathsf{DispAlg}\,\{\mathsf{NatSig}\}\,(X,\ zero,\ suc)|\,\{\bullet\}\,\mathsf{tt} \equiv$$
$$(X^D \quad : \mathsf{Ty}_{\mathbb{C}}\,(\bullet \triangleright n : |X|\,\mathsf{tt}))$$
$$\times\,(zero^D : \mathsf{Tm}_{\mathbb{C}}\,\bullet\,(X^D[n \mapsto |zero|\,\mathsf{tt}]))$$
$$\times\,(suc^D \quad : \mathsf{Tm}_{\mathbb{C}}\,(\bullet \triangleright n : |X|\,\mathsf{tt} \triangleright n^D : X^D[n \mapsto |zero|\,\mathsf{tt}])\,(X^D[n \mapsto (|suc|\,\mathsf{tt})[n \mapsto n]]))$$

To explain $(|\mathsf{suc}|\,\mathsf{tt})[n \mapsto n]$): we have $\mathsf{suc}\,n$ in 2LTT, where n is an inner variable, and in the presheaf model inner variables become actual variables in the inner theory. Hence, we map the n which suc depends on to the concrete n in the context.

We can also interpret displayed algebras in finite product categories:

$$|\mathsf{DispAlg}\,\{\mathsf{NatSig}\}\,(X,\ zero,\ suc)|\,\{\bullet\}\,\mathsf{tt} \equiv$$
$$(X^D \quad : |\mathbb{C}|)$$
$$\times\,(zero^D : \mathbb{C}(\bullet,\ X^D))$$
$$\times\,(suc^D \quad : \mathbb{C}(\bullet \otimes |X|\,\mathsf{tt} \otimes X^D,\ X^D))$$

While displayed algebras in cwfs can be used as bundles of induction motives and methods, in finite product categories they are argument bundles to *primitive recursion*; this is sometimes also called a *paramorphism* [MFP91]. In an internal syntax, the type of primitive recursion for natural numbers could be written more compactly as:

$$\mathsf{primrec} : (X : \mathsf{Set}) \to X \to (\mathsf{Nat} \to X \to X) \to \mathsf{Nat} \to X$$

This is not the same thing as the usual recursion principle (corresponding to weak initiality), because of the extra dependency on Nat in the method for successors.

3.5.5   Sections

Sections are analogous to morphisms. We again have a choice between weak and strict preservation; below we have weak preservation.

$$
\begin{aligned}
{-}^S : (A : \mathsf{Ty}) &\{X\ X^D\}(X^S : (x : \mathsf{Tm}_0\ X) \to \mathsf{Tm}_0\ (X^D\ x)) \\
&\to (\alpha : A^A\ X) \to A^D\ X^D\ \alpha \to \mathsf{Set} \\
\iota^S \qquad X^S\ \alpha\ &\alpha^D :\equiv \mathsf{Tm}_0\ (X^S\ \alpha = \alpha^D) \\
(\iota \to A)^S\ X^S\ \alpha\ &\alpha^D :\equiv (x : \mathsf{Tm}_0\ X) \to A^S\ X^S\ (\alpha\ x)\,(\alpha^D\ (X^S\ x))
\end{aligned}
$$

3.5.6   Term Algebras

For term algebras, we need to assume a bit more in the inner theory. For starters, it has to support the theory of signatures. In order to avoid name clashes down the line, we use SigTy_0 to refer to signature types, and SigTm_0 for terms. That is, we have

$$
\begin{aligned}
\mathsf{SigTy}_0 \ &: \mathsf{Ty}_0 \\
\mathsf{Con}_0 \ \ &: \mathsf{Ty}_0 \\
\mathsf{Var}_0 \ \ &: \mathsf{Tm}_0\ \mathsf{Con}_0 \to \mathsf{Tm}_0\ \mathsf{SigTy}_0 \to \mathsf{Ty}_0 \\
\mathsf{SigTm}_0 \ &: \mathsf{Tm}_0\ \mathsf{Con}_0 \to \mathsf{Tm}_0\ \mathsf{SigTy}_0 \to \mathsf{Ty}_0 \\
\mathsf{Sub}_0 \ \ &: \mathsf{Tm}_0\ \mathsf{Con}_0 \to \mathsf{Tm}_0\ \mathsf{Con}_0 \to \mathsf{Ty}_0
\end{aligned}
$$

together with all constructors and induction principles. We also assume inner Π-types, because we previously defined Sub using functions.

Remark. In the current section we are liberal with making assumptions in the inner theory. Strictly speaking, if we only want to construct term algebras, we don't need to assume signature induction principles. Also, we will see in Section **??** that inner ToS induction is not necessary to show weak initiality of term algebras; weaker assumptions suffice. In this section, we omit these generalizations, as the goal is to redo the constructions of Chapter 2 without making essential changes.

We still have ToS in the outer layer. To make the naming scheme consistent, we shall write outer ToS types as SigTy_1, SigTm_1, Con_1, Var_1 and Sub_1. We have conversion functions from the outer ToS to the inner ToS:

Definition 38. We have the following **lowering** functions which preserve all structure.

$$
\begin{aligned}
\downarrow &: \mathsf{SigTy}_1 && \to \mathsf{Tm}_0\,\mathsf{SigTy}_0 \\
\downarrow &: \mathsf{Con}_1 && \to \mathsf{Tm}_0\,\mathsf{Con}_0 \\
\downarrow &: \mathsf{Var}_1\,\Gamma\,A && \to \mathsf{Tm}_0\,(\mathsf{Var}_0\,(\downarrow\Gamma)\,(\downarrow A)) \\
\downarrow &: \mathsf{SigTm}_1\,\Gamma\,A && \to \mathsf{Tm}_0\,(\mathsf{SigTm}_0\,(\downarrow\Gamma)\,(\downarrow A)) \\
\downarrow &: \mathsf{Sub}_1\,\Gamma\,\Delta && \to \mathsf{Tm}_0\,(\mathsf{Sub}_0\,(\downarrow\Gamma)\,(\downarrow\delta))
\end{aligned}
$$

These functions are called "lifting" in the context of multi-stage programming; see e.g. the Lift typeclass in Haskell [PWK19]. There, like here, the point is to build object-language terms from meta-level ("compile-time") values.

Lowering is straightforward to define for types, contexts, variables and terms, but there is a bit of a complication for Sub. Unfolding the definitions, we need to map from $\{A\} \to \mathsf{Var}_1\,\Delta\,A \to \mathsf{SigTm}_1\,\Gamma\,A$ to $\mathsf{Tm}_0\,(\{A\} \to \mathsf{Var}_0\,(\downarrow\Delta)\,A \to \mathsf{SigTm}_0\,(\downarrow\Gamma)\,A)$. It might appear problematic that we have types and variables in *negative* position, because we can't map inner types/variables to outer ones. Fortunately, $\mathsf{Sub}_1\,\Gamma\,\Delta$ is isomorphic to a finite product type, and we can lower a finite product component-wise.

Concretely, we define lowering by induction on Δ, while making use of a case splitting operation for Var_0. We use an informal case operation below, which can be defined using inner induction. Note that since $\mathsf{Var}_0\,\bullet\,A$ is an empty type, case splitting on it behaves like elimination for the empty type.

$$
\begin{aligned}
\downarrow_\Delta &: \mathsf{Sub}_1\,\Gamma\,\Delta \to \mathsf{Tm}_0\,(\mathsf{Sub}_0\,(\downarrow\Gamma)\,(\downarrow\Delta)) \\
\downarrow_\bullet\quad &\sigma :\equiv \lambda\,\{A\}\,(x : \mathsf{Var}_0\,\bullet\,A).\,\mathsf{case}\,x\,\mathsf{of}\,() \\
\downarrow_{\Delta\triangleright B}\,&\sigma :\equiv \lambda\,\{A\}\,(x : \mathsf{Var}_1\,(\downarrow\Delta \triangleright \downarrow B)\,A).\,\mathsf{case}\,x\,\mathsf{of} \\
&\quad\quad \mathsf{vz}\ \ \to\ \downarrow(\sigma\,\mathsf{vz}) \\
&\quad\quad \mathsf{vs}\,x \to\ \downarrow_\Delta\,(\sigma \circ \mathsf{vs})\,x
\end{aligned}
$$

In general, for finite A type, functions of the form $A \to \mathsf{Tm}_0\,B$ can be represented as inner types up to isomorphism; they can be viewed as finite products of terms.

Remark. For infinite A this does not work anymore in our system. In [ACKS19], the assumption that this still works with $A \equiv \mathsf{Nat}_1$ is an important axiom ("cofibrancy of Nat_1") which makes it possible to embed higher categorical structures in 2LTT. From the metaprogramming perspective, cofibrancy of Nat_1

implies that the inner theory is *infinitary*, since we can form inner terms from infinite collections of inner terms. We don't assume this axiom in 2LTT, although we will consider infinitary (object) type theories in Chapters 4 and 5.

We continue to the definition of term algebras. We fix an $\Omega : \mathsf{Con}_1$, and define $\mathsf{T} : \mathsf{Ty}_0$ as $\mathsf{SigTm}_0 \, (\downarrow \Omega) \, \iota$.

$$-^T : (A : \mathsf{SigTy}_1) \to \mathsf{Tm}_0 \, (\mathsf{SigTm}_0 \, (\downarrow \Omega) \, (\downarrow A)) \to A^A \, \mathsf{T}$$
$$\iota^T \qquad t :\equiv t$$
$$(\iota \to A)^T \, t :\equiv \lambda \, u. \, A^T \, (\mathsf{app} \, t \, u)$$

$$-^T : (\Gamma : \mathsf{Con}_1) \to \mathsf{Sub}_1 \, \Omega \, \Gamma \to \Gamma^A \, \mathsf{T}$$
$$\Gamma^T \, \nu \, \{A\} \, x :\equiv A^T \, (\downarrow (\nu \, x))$$

$$\mathsf{TmAlg}_\Omega : \mathsf{Alg} \, \Omega$$
$$\mathsf{TmAlg}_\Omega :\equiv \Omega^T \, \Omega \, \mathsf{id}$$

We omit the $-^T$ interpretation for terms and substitutions for now, as they require a bit more setup, and they are not needed just for term algebras.

3.5.7 Weak Initiality

Recall from Section 2.3.1 that recursion is implemented using the $-^A$ interpretation of terms. Since terms are now in the inner theory, we need to define an inner version of the same interpretation. We need to compute types by inner induction, so we additionally assume a Russell-style inner U_0 universe. The Russell style means that we may freely coerce between $\mathsf{Tm}_0 \, \mathsf{U}_0$ and Ty_0. The following are defined the same way as $-^A$ before.

$$-^A : \mathsf{Tm}_0 \, (\mathsf{SigTy}_0 \to \mathsf{U}_0 \to \mathsf{U}_0)$$
$$-^A : \mathsf{Tm}_0 \, (\mathsf{Con}_0 \to \mathsf{U}_0 \to \mathsf{U}_0)$$
$$-^A : \mathsf{Tm}_0 \, (\mathsf{SigTm}_0 \, \Gamma \, A \to \{X : \mathsf{U}_0\} \to \Gamma^A \, X \to A^A \, X)$$
$$-^A : \mathsf{Tm}_0 \, (\mathsf{Sub}_0 \, \Gamma \, \Delta \to \{X : \mathsf{U}_0\} \to \Gamma^A \, X \to \Delta^A \, X)$$

Since lowering preserves all structure, and $-^A$ is defined in the same way in both the inner and outer theories, lowering is compatible with $-^A$ in the following way.

Lemma 4. Assume $A : \mathsf{SigTy}_1$, $\Gamma : \mathsf{Con}_1$, $X : \mathsf{Ty}_0$, $\gamma : \Gamma^A X$ and $t : \mathsf{SigTm}_1 \Gamma A$. We have the following:

- $(A^A_\rightarrow, A^A_\leftarrow) : \mathsf{Tm}_0 ((\downarrow A)^A X) \simeq A^A X$

- $(\Gamma^A_\rightarrow, \Gamma^A_\leftarrow) : \mathsf{Tm}_0 ((\downarrow\Gamma)^A X) \simeq \Gamma^A X$

- $t^A \gamma \equiv A^A_\rightarrow ((\downarrow t)^A (\Gamma^A_\leftarrow \gamma))$

Proof. By induction on Γ, A and t. $\qquad\qquad\qquad\qquad\qquad\qquad$ □

We construct recursors now, yielding strict algebra morphisms. We assume $(X, \omega) : \mathsf{Alg}\,\Omega$. Recall that $\omega : \Omega^A X$, thus $\Omega^A_\leftarrow \omega : \mathsf{Tm}_0 ((\downarrow\Omega)^A X)$. We define $\mathsf{R} : \mathsf{Tm}_0 \mathsf{T} \rightarrow \mathsf{Tm}_0 X$ as $\mathsf{R}\,t :\equiv t^A (\Omega^A_\leftarrow \omega)$.

$$-^R : (A : \mathsf{SigTy}_1)(t : \mathsf{Tm}_0 (\mathsf{SigTm}_0 (\downarrow\Omega) (\downarrow A))) \rightarrow A^M \mathsf{R} (A^T t) (\mathsf{A}^A_\rightarrow (t^A (\Omega^A_\leftarrow \omega)))$$
$$\iota^R \qquad t : t^A (\Omega^A_\leftarrow \omega) \equiv \iota^A_\rightarrow (t^A (\Omega^A_\leftarrow \omega))$$
$$(\iota \rightarrow A)^R t :\equiv \lambda u.\, A^R (\mathsf{app}\,t\,u)$$

$$-^R : (\Gamma : \mathsf{Con}_1)(\nu : \mathsf{Sub}_1 \Omega\,\Gamma) \rightarrow \Gamma^M \mathsf{R} (\Gamma^T \nu) (\nu^A \omega)$$
$$\Gamma^R \nu \{A\}\, x :\equiv A^R (\downarrow(\nu\,x))$$

In the proof obligation for $t^A (\Omega^A_\leftarrow \omega) \equiv \iota^A_\rightarrow (t^A (\Omega^A_\leftarrow \omega))$, $\iota^A_\rightarrow$ computes to the identity function; note that $\iota^A_\rightarrow : \mathsf{Tm}_0 X \rightarrow \mathsf{Tm}_0 X$. Hence the equality becomes reflexive.

In $\Gamma^R \nu \{A\}\, x :\equiv A^R (\downarrow(\nu\,x))$, we have that

$$A^R \downarrow(\nu\,x) : A^M \mathsf{R} (A^T (\downarrow(\nu\,x))) (A^A_\rightarrow (\downarrow(\nu\,x)^A (\Omega^A_\leftarrow \omega)))$$

Hence by Lemma 4, we have

$$A^R \downarrow(\nu\,x) : A^M \mathsf{R} (A^T (\downarrow(\nu\,x))) ((\nu\,x)^A \omega)$$

Hence, by the definition of $-^A$ for substitutions:

$$A^R \downarrow(\nu\,x) : A^M \mathsf{R} (A^T (\downarrow(\nu\,x))) (\nu^A \omega\,x)$$

Which is exactly what is required when we unfold the expected return type:

$$-^R : (\Gamma : \mathsf{Con}_1)(\nu : \mathsf{Sub}_1 \Omega\,\Gamma) \rightarrow \Gamma^M \mathsf{R} (\Gamma^T \nu) (\nu^A \omega)$$
$$-^R : (\Gamma : \mathsf{Con}_1)(\nu : \mathsf{Sub}_1 \Omega\,\Gamma) \rightarrow \{A\}(x : \mathsf{Var}_1 \Gamma\,A) \rightarrow A^M \mathsf{R} (A^T (\downarrow(\nu\,x))) (\nu^A \omega\,x)$$

The recursor is defined the same way as in Definition 10:

$$\mathsf{Rec}_\Omega : (alg : \mathsf{Alg}\,\Omega) \to \mathsf{Mor}\,\mathsf{TmAlg}_\Omega\,alg$$
$$\mathsf{Rec}_\Omega\,(X,\,\omega) :\equiv (\mathsf{R},\,\Omega^R\,\Omega\,\mathsf{id})$$

3.5.8 Induction

For induction, we need to additionally define $-^D$ in the inner layer.

$$-^D : \mathsf{Tm}_0\,((A : \mathsf{SigTy}_0)\{X\} \to (\mathsf{Tm}_0\,X \to \mathsf{U}_0) \to A^A\,X \to \mathsf{U}_0)$$
$$-^D : \mathsf{Tm}_0\,((\Gamma : \mathsf{Con}_0)\{X\} \to (\mathsf{Tm}_0\,X \to \mathsf{U}_0) \to \Gamma^A\,X \to \mathsf{U}_0)$$
$$-^D : \mathsf{Tm}_0\,((t : \mathsf{SigTm}_0\,\Gamma\,A) \to \Gamma^D\,X^D\,\gamma \to A^D\,X^D\,(t^A\,\gamma))$$
$$-^D : \mathsf{Tm}_0\,((\sigma : \mathsf{Sub}_0\,\Gamma\,\Delta) \to \Gamma^D\,X^D\,\gamma \to \Delta^D\,X^D\,(\sigma^A\,\gamma))$$

Lemma 5. We have again compatibility of lowering with $-^D$. Assuming $(X,\,\gamma)$: $\mathsf{Alg}\,\Gamma$, $(X^D,\,\gamma^D)$: $\mathsf{DispAlg}\,(X,\,\gamma)$, $t : \mathsf{SigTm}_1\,\Gamma\,A$, and $\alpha : A^A\,X$, we have

- $(\mathsf{A}^D_\to,\,\mathsf{A}^D_\leftarrow) : \mathsf{Tm}_0\,((\downarrow A)^D\,X^D\,(\mathsf{A}^A_\leftarrow\,\alpha)) \simeq \mathsf{A}^D\,X^D\,\alpha$

- $(\Gamma^D_\to,\,\Gamma^D_\leftarrow) : \mathsf{Tm}_0\,((\downarrow\Gamma)^D\,X^D\,(\Gamma^A_\leftarrow\,\gamma)) \simeq \Gamma^D\,X^D\,\gamma$

- $t^D\,\gamma^D \equiv A^D_\to\,((\downarrow t)^D\,(\Gamma^D_\leftarrow\,\gamma^D))$

The equation for $t^D\,\gamma^D$ is well-typed because of the term equation in Lemma 4.

Proof. Again by induction on Γ, A and t. $\qquad\qquad\qquad\qquad\square$

We also need to extend $-^T$ with action on terms. Note that we return an inner equality, since we can only compute such equality by induction on the inner term input:

$$-^T : (t : \mathsf{SigTm}_0\,(\downarrow\Gamma)\,(\downarrow A))(\nu : \mathsf{Sub}_1\,\Omega\,\Gamma) \to \mathsf{Tm}_0\,(A^A_\leftarrow\,(A^T\,(t[\downarrow\nu])) = t^A\,(\Gamma^A_\leftarrow\,\nu))$$

We assume $(X^D,\,\omega^D)$: $\mathsf{DispAlg}\,\mathsf{TmAlg}_\Omega$, and define elimination as follows:

$$\mathsf{E} : (t : \mathsf{Tm}_0\,\mathsf{T}) \to \mathsf{Tm}_0\,(X^D\,t)$$
$$\mathsf{E}\,t :\equiv t^D\,(\Omega^D_\leftarrow\,\omega^D)$$

This definition is well-typed only up to $t^T\,\mathsf{id} : \mathsf{Tm}_0\,(t = t^A\,(\Omega^A_\leftarrow\,(\Omega^T\,\Omega\,\mathsf{id})))$. Since $t^T\,\mathsf{id}$ is an inner equality, in a fully formal intensional presentation we would have to write an explicit transport in the definition.

We shall skip the remainder of the eliminator construction; it goes the same way as in Definition 11. Intuitively, this is possible since the inner theory has all necessary features to reproduce the eliminator construction, and lowering preserves all structure.

Since t^T yields inner equations, this implies that the displayed algebra sections returned by the eliminator are *weak sections*, i.e. they contain β-rules expressed in inner equalities.

3.6   Discussion

3.6.1   Evaluation

Let's review how 2LTT addresses the shortcomings that we mentioned in Section 3.1.

Generalized semantics. For this purpose, there is minimal technical overhead to using 2LTT; we only need to sprinkle Ty_0 and Tm_0 around a bit, and we get semantics internally to cwfs (including all finite product categories).

Term algebra constructions. Here, we had to deal with significantly more noise, because of the necessary lowerings and their preservation isomorphisms. However, this overhead should be compared to reasoning about object-level definitional equality in a deeply embedded way, which entails explicitly manipulating abstract syntax and its substitutions and weakenings. Compared to that, 2LTT is tremendously easier. One could also argue that lowering is an entirely mechanical affair, and we can just omit most of it if we are comfortable enough with the formalism.

3.6.2   Recursor vs. Eliminator Construction

The essential extra detail that we had to handle in Section 3.5 was the choice between strict and weak equations. This choice brings along further implementation constraints.

Strict equations are stronger as assumptions, because they represent definitional equality of inner terms. However, we can only produce strict equations by eliminating from outer types. Hence, if we aim to output strict equations, we

have to assume every dependency in the outer theory, which in turn may require using lowering.

Weak equations are easier to produce: strict equality implies weak equality, plus we can prove weak equality by inner induction. But we cannot use weak equality to transport outer values.

In the recursor construction, we produced strict β-rules, which trivially imply weak β-rules as well. In contrast, the eliminator construction relies essentially on equations produced by $-^T$ for inner terms. Can we somehow get strict equations from $-^T$? This does not seem possible, at least without changing the approach in a major way: terms must be inner, or else we have no hope of inner recursion/induction. And if terms are inner, then E must act on arbitrary inner terms, hence $-^T$ must do so too.

It would be interesting to check if there is a possible alternative formulation of term algebras and constructed eliminators, which makes it possible to get strict eliminators. Looking back to Section 2.4.2, it appears that by using spine neutral terms, we get strict elimination for each signature. However, this relies on the Agda implementation of pattern matching and structurally recursive definitions, so it would require more work to translate these definitions to 2LTT in a generic way.

CHAPTER 4

Finitary Quotient Inductive-Inductive Types

In this chapter we bump the expressive power of signatures by a large margin, and also substantially extend the semantics. However, we keep the basic approach the same; indeed its advantages become more apparent with the more sophisticated signatures.

4.1 Theory of Signatures

Signatures are once again given by contexts of a type theory, but now it is a dependent type theory, given as a cwf with certain type formers, in the style of Section 3.2.

Metatheory and notation. We work in 2LTT with Ty_0 and Tm_0. We make the following assumptions. Later, we may assume more type formers at either levels as needed.

- Ty_0 is closed under $\top$, Σ and extensional identity $- = -$. The inner identity reflects the outer one.

- The outer identity $- \equiv -$ is also extensional; it supports UIP, function extensionality, and it reflects strict equality in some unspeficied metatheory outside 2LTT. This reflection is used to justify omitting transports along $- \equiv -$ in our notation.

In the following we specify models of the theory of *finitary quotient inductive-inductive signatures.* The names involved are a bit of a mouthful, so we abbreviate "finitary quotient inductive-inductive" as FQII, and like before, we abbreviate

"theory of signatures" as ToS. In this chapter, by signature we mean an FQII signature unless otherwise specified.

Definition 39. A **model of the theory of signatures** consists of the following.

- A **cwf** with underlying sets Con, Sub, Ty and Tm, all returning in the outer Set universe of 2LTT.

- A **Tarski-style universe** U with decoding El.

- An **extensional identity type** $\mathsf{Id} : \mathsf{Tm}\,\Gamma\,A \to \mathsf{Tm}\,\Gamma\,A \to \mathsf{Ty}\,\Gamma$, specified by $(\mathsf{reflect}, \mathsf{refl}) : \mathsf{Tm}\,\Gamma\,(\mathsf{Id}\,t\,u) \simeq (t \equiv u)$.

- An **inductive function type** $\Pi : (a : \mathsf{Tm}\,\Gamma\,\mathsf{U}) \to \mathsf{Ty}\,(\Gamma \triangleright \mathsf{El}\,a) \to \mathsf{Ty}\,\Gamma$, specified by $(\mathsf{app}, \mathsf{lam}) : \mathsf{Tm}\,\Gamma\,(\Pi\,a\,B) \simeq \mathsf{Tm}\,(\Gamma \triangleright \mathsf{El}\,a)\,B$.

- An **external function type** $\Pi^{\mathsf{ext}} : (A : \mathsf{Ty}_0) \to (A \to \mathsf{Ty}\,\Gamma) \to \mathsf{Ty}\,\Gamma$, specified by $(\mathsf{app}^{\mathsf{ext}}, \mathsf{lam}^{\mathsf{ext}}) : \mathsf{Tm}\,\Gamma\,(\Pi^{\mathsf{ext}}\,A\,B) \simeq ((x : A) \to \mathsf{Tm}\,\Gamma\,(B\,x))$.

At this point we only have a notion of model for ToS, but as we will see in Chapter 5, ToS is also an algebraic theory, more specifically an infinitary QII one. It is infinitary because Π^{ext} and $\mathsf{lam}^{\mathsf{ext}}$ allow branching over elements of arbitrary $A : \mathsf{Ty}_0$ types.

Because of the algebraic character of ToS, there is a category of ToS models where morphisms strictly preserve all structure, and the initial model corresponds to the syntax. In this chapter, we just assume that the ToS syntax exists, and leave metatheoretical matters to Chapter 5.

Definition 40. An FQII **signature** is an element of Con in the syntax of ToS.

We review several example signatures in the following, using progressively more ToS type formers. We also introduce progressively more compact notation for signatures. As a rule of thumb, we shall use compact notation for larger and more complex signatures, but we shall be more explicit when we specify models of ToS later in this chapter.

Example 10. Simple inductive signatures can be evidently expressed using U and Π. By adding a single U to the signature, we introduce the inductive sort.

$$\mathsf{NatSig} :\equiv \bullet \triangleright (N : \mathsf{U}) \triangleright (zero : \mathsf{El}\,N) \triangleright (suc : \Pi(n : N)(\mathsf{El}\,N))$$
$$\mathsf{TreeSig} :\equiv \bullet \triangleright (T : \mathsf{U}) \triangleright (leaf : \mathsf{El}\,T) \triangleright (node : \Pi(t_1 : T)(\Pi(t_2 : T)(\mathsf{El}\,T)))$$

Observe that the domains in Π are terms with type U, while the codomains are proper types.

Notation 12. We write non-dependent function types in ToS as follows.

- $a \Rightarrow B$ for $\Pi \, (\_ : a) \, B$.

- $A \Rightarrow^{ext} B$ for $\Pi^{\mathsf{ext}} \, A \, (\lambda \, \_ . \, B)$.

Using this notation, we may write $suc : N \Rightarrow \mathsf{El}\, N$ and $node : T \Rightarrow T \Rightarrow \mathsf{El}\, T$.

Notation 13. The "categorical" application app with explicit substitutions is a bit inconvenient. Instead, we simply write whitespace for Π and Π^{ext} application:

$$t \, u :\equiv (\mathsf{app}\, t)[\mathsf{id},\, u]$$
$$t \, u :\equiv (\mathsf{app}^{\mathsf{ext}}\, t) \, u$$

Example 11. We may have any number of sorts by adding more U to the signatures. Moreover, sorts can be indexed over previous sorts. Hence, using only U, El and Π, we can express any closed inductive-inductive type [**?**]. The following fragment of the the signature for categories is such:

$$\bullet \triangleright (Obj : \mathsf{U}) \triangleright (Hom : Obj \Rightarrow Obj \Rightarrow \mathsf{U}) \triangleright (id : \Pi(i : Obj) \, (\mathsf{El}\, (Hom \, i \, i)))n$$

These inductive-inductive signatures are more flexible than those in prior works [**?**], since we allow type constructors (sorts) and point constructors to be arbitrarily mixed (as opposed to mandating that sorts are declared first). For example:

$$\bullet \triangleright (A : \mathsf{U}) \triangleright (a : \mathsf{El}\, A) \triangleright (B : A \Rightarrow \mathsf{U}) \triangleright (C : B \, a \Rightarrow \mathsf{U})$$

Here C is indexed over $B \, a$, where a is a point constructor of a, so a sort specification mentions a point constructor.

Example 12. Id lets us add equations to signatures. With this, we can write

down the full signature for categories:

- $\triangleright (Obj \quad : \mathsf{U})$
  $\triangleright (Hom : Obj \Rightarrow Obj \Rightarrow \mathsf{U})$
  $\triangleright (id \quad : \Pi(i : Obj)(\mathsf{El}(Hom\,i\,i)))$
  $\triangleright (comp : \Pi(i\,j\,k : Obj)(Hom\,j\,k \Rightarrow Hom\,i\,j \Rightarrow \mathsf{El}(Hom\,i\,k)))$
  $\triangleright (idr \quad : \Pi(i\,j : Obj)(f : Hom\,i\,j)(\mathsf{Id}(comp\,i\,i\,j\,f\,(id\,i))\,f))$
  $\triangleright (idl \quad : \Pi(i\,j : Obj)(f : Hom\,i\,j)(\mathsf{Id}(comp\,i\,j\,j\,(id\,j)\,f)\,f))$
  $\triangleright (assoc : \Pi(i\,j\,k\,l : Obj)(f : Hom\,j\,l)(g : Hom\,j\,k)(h : Hom\,i\,j)$
  $\qquad\qquad (\mathsf{Id}(comp\,i\,j\,l\,(comp\,j\,k\,l\,f\,g)\,h)\,(comp\,i\,k\,l\,f\,(comp\,i\,j\,k\,g\,h))$

Now, this is already rather hard to read, even together with the compressed notation for multiple Π binders.

Notation 14. For more complex signatures, we may entirely switch to an internal notation, where we mostly reuse the conventions in the metatheories. We use $(x : a) \to B$ for inductive functions, $(x : A) \to^{ext} B$ for external functions, but we still write Id for the identity type and make U and El explicit. In this notation, a signature is just a listing of binders. The category signature becomes the following:

$$
\begin{aligned}
Obj \quad &: \mathsf{U} \\
Hom \quad &: Obj \to Obj \to \mathsf{U} \\
id \quad &: \mathsf{El}(Hom\,i\,i) \\
-\circ- \quad &: Hom\,j\,k \to Hom\,i\,j \to \mathsf{El}(Hom\,i\,k) \\
idr \quad &: \mathsf{Id}(f \circ id)\,f \\
idl \quad &: \mathsf{Id}(id \circ f)\,f \\
assoc \quad &: \mathsf{Id}(f \circ (g \circ h))\,((f \circ g) \circ h)
\end{aligned}
$$

Example 13. The external function type makes it possible to reference inner types (in 2LTT) in signatures. Here "external" is meant relative to a given signature, and refers to types and inhabitants which are not introduced inside a signature. For example, we give a signature for lists by assuming $A : \mathsf{Ty}_0$ for the

(external) type of list elements:

$$
\begin{aligned}
List\ &: \mathsf{U} \\
nil\ \ &: \mathsf{El}\ List \\
cons\ &: A \to^{ext} List \to \mathsf{El}\ List
\end{aligned}
$$

Hence, "parameters" are always assumptions made in the metatheory. We can also *index* sorts by external values. Let's specify length-indexed vectors now; we keep the $A : \mathsf{Ty}_0$ assumption, but also assume that Ty_0 has natural numbers, with $\mathsf{Nat}_0 : \mathsf{Ty}_0$, zero_0 and suc_0.

$$
\begin{aligned}
Vec\ \ &: \mathsf{Nat}_0 \to^{ext} \mathsf{U} \\
nil\ \ \ &: \mathsf{El}\,(\,Vec\,\mathsf{zero}_0) \\
cons\ &: (n : \mathsf{Nat}_0) \to^{ext} A \to^{ext} Vec\,n \to \mathsf{El}\,(\,Vec\,(\mathsf{suc}_0\,n))
\end{aligned}
$$

Example 14. We can also introduce *sort equations* using Id: this means equating terms of U, i.e. inductively specified sets. This is useful for specifying certain strict type formers. For example, a signature for cwfs can be extended with a specification for strict constant families.

$$
\begin{aligned}
Con\ \ &: \mathsf{U} \\
Sub\ \ &: Con \to Con \to \mathsf{U} \\
Ty\ \ \ &: Con \to \mathsf{U} \\
Tm\ \ &: (\Gamma : Con) \to Ty\,\Gamma \to \mathsf{U} \\
&\dots \\
K\ \ \ \ &: Con \to \{\Gamma : Con\} \to \mathsf{El}\,(\,Ty\,\Gamma) \\
K_{spec}\ &: \mathsf{Id}\,(\,Tm\,\Gamma\,(K\,\Delta))\,(Sub\,\Gamma\,\Delta)
\end{aligned}
$$

The equation for Russell-style universes is likewise a sort equation:

$$
\begin{aligned}
Univ\ \ \ &: \mathsf{El}\,(\,Ty\,\Gamma) \\
Russell\ &: \mathsf{Id}\,(\,Tm\,\Gamma\,Univ)\,(\,Ty\,\Gamma)
\end{aligned}
$$

Example 15. As we mentioned in Definition 25, there is a signature for presheaves, so let's look at that now. Assume a category $\mathbb{C}$ in the inner theory; this means

that objects and morphisms of $\mathbb{C}$ are in Ty_0.

$$
\begin{aligned}
Obj \quad &: |\mathbb{C}| \to^{ext} \mathsf{U} \\
Hom \quad &: \mathbb{C}(i,\,j) \to^{ext} Obj\,j \to \mathsf{El}\,(Obj\,i) \\
Hom_{\mathsf{id}} &: \mathsf{Id}\,(Hom\,\mathsf{id}\,x)\,x \\
Hom_{\circ} \;\; &: \mathsf{Id}\,(Hom\,(f \circ g)\,x)\,(Hom\,f\,(Hom\,g\,x))
\end{aligned}
$$

We depart from the sugary naming scheme in Definition 25, and name the action on objects Obj and the action on morphisms Hom. When we give semantics to this signature in Section **??**, we will get as algebras presheaves from $\mathbb{C}$ to the category of inner types. That category has elements of Ty_0 as objects and $\mathsf{Tm}_0\,A \to \mathsf{Tm}_0\,B$ functions as morphisms.

Strict positivity. Only strictly positive signatures are expressible. Similarly to the case with simple signatures, there is no way to abstract over a inductive functions, since inductive function domains are in U, and U has no type formers at all. With Π^{ext}, we can abstract over functions, but only on ones which are external to a signature and don't depend on internally specified constructors.

4.2 Semantics

4.2.1 Overview

For simple signatures, we only gave semantics in enough detail so that notions of recursion and induction could be recovered. We aim to do more now. For each signature, we would like to have

1. A *category* of algebras, with homomorphisms as morphisms.

2. A notion of induction, which requires a notion of dependent algebras.

3. A proof that for algebras, initiality is equivalent to supporting induction.

We do this by creating a model of ToS where contexts (signatures) are categories with certain extra structure and substitutions are structure-preserving functors. Then, ToS signatures can be interpreted in this model, using the initiality of ToS syntax (i.e. the recursor).

Our semantics has a type-theoretic flavor, which is inspired by the cubical set model of Martin-Löf type theory by Bezem et al. [**?**]. The idea is to avoid strictness issues by starting from basic ingredients which are already strict enough. Hence, instead of modeling ToS types as certain slices and substitution by pullback, we model types as displayed categories with extra structure, which naturally support strict reindexing/substitution.

We make a similar choice in the interpretation of signatures themselves: we use structured cwfs of algebras, where types correspond to displayed algebras. This choice is in contrast to having finitely complete categories of algebras. Preliminarily, the reason is that "native" displayed algebras and sections allow us to compute induction principles strictly as one would write in a type theory. In fact, in this chapter we recover exactly the same semantics for simple signatures that we already specified.

In contrast, displayed algebras are a derived notion in finitely complete categories, and the induction principles would be only up to isomorphism. This issue is perhaps not relevant from a purely categorical perspective, but we are concerned with eventually implementing QIITs in proof assistants. If we don't compute induction principles here in an exact way, we don't get them from anywhere else.

4.2.2 Separate vs. Bundled Models

Previously, we defined $-^A$, $-^M$, $-^D$ and $-^S$ interpretations of signatures separately, by doing induction anew for each one. Formally, this amounts to giving a plain model of ToS in order to define $-^A$, but then giving three *displayed* models of ToS to specify the other interpretations, because they sometimes need to refer to the recursors or eliminators of other interpretations.

For example, $-^A : \mathsf{Con} \to \mathsf{Set}$ while $-^D : (\Gamma : \mathsf{Con}) \to \Gamma^A \to \mathsf{Set}$, so displayed algebras already refer to $-^A$, which is part of the recursor for the corresponding model.

However, this piecewise style can be avoided: we can give a single non-displayed model which packs everything in a Σ-type, yielding just one inter-

pretation function for signatures. Let's call that function $-^M$ now:

$$
\begin{aligned}
-^M : \mathsf{Con} \to\, & (A\ : \mathsf{Set}) \\
& \times\ (M : A \to A \to \mathsf{Set}) \\
& \times\ (D\ : A \to \mathsf{Set}) \\
& \times\ (S\ \ : (a : A) \to D\, a \to \mathsf{Set})
\end{aligned}
$$

Note that it is often not possible to merge multiple recursors/eliminators by packing models together. For example, addition on natural numbers is defined by recursion, and so is multiplication; but since multiplication calls addition in an iterated fashion, it is not possible to define both operations by a single algebra. Nevertheless, merging does work in our case. We will, in fact, get a formal vocabulary for merging models (and manipulating them in other ways) from the semantics of ToS itself.

In simple cases, and in Agda, the piecewise style is convenient, since we don't have to deal with Σ-s. However, for larger models, important organizing principles may become more apparent if we bundle things together.

In the following, we shall define a model $\mathbf{M} : \mathsf{ToS}$ such that its Con component is a bundle containing all A, M, D, S components, plus a number of additional components. We present the components of $\mathbf{M}$ in the same order as in Definition 39. There is significant overlap in names and notations, so we use **bold** font to disambiguate components of $\mathbf{M}$ from components of other structures. For example, we use $\boldsymbol{\sigma} : \mathbf{Sub}\,\boldsymbol{\Gamma}\,\boldsymbol{\Delta}$ to denote a substitution in $\mathbf{M}$, while there could be a Sub-named components in other structures under consideration.

4.2.3 Finite Limit Cwfs

We define $\mathbf{Con}$: Set as the type of finite limit cwfs. Recall that this specifies the objects of the underlying cwf of $\mathbf{M}$. In the following we specify flcwfs and describe some internal constructions.

Definition 41. We define flcwf : Set as an iterated Σ-type with the following components:

1. A cwf with Con, Sub, Ty, Tm all returning in Set. *Remark:* this implies that flcwf : Set is in a larger universe than all of these internal components. We continue to elide universe sizing details.

2. Σ-types.

3. Extensional identity type Id with refl and $\mathsf{reflect}$.

4. Strict constant families K.

Definition 42. We abbreviate the additional structure on cwfs consisting of Σ, Id and K as **fl-structure**.

We recover previous concepts as follows. Assuming Γ signature, we get an flcwf by interpreting Γ in $\mathbf{M}$. In that flcwf we have

- Con as the type of algebras.

- Sub as the type of algebra morphisms.

- Ty as the type of displayed algebras.

- Tm as the type of displayed algebra sections.

From this, notions of initiality and induction are apparent as well. Initiality is the usual categorical notion. For induction, assuming $\boldsymbol{\Gamma} : \mathbf{Con}$, we have the following predicate:

$$\mathsf{Inductive} : \mathsf{Con}_{\boldsymbol{\Gamma}} \to \mathsf{Set}$$
$$\mathsf{Inductive}\,\Gamma :\equiv (A : \mathsf{Ty}_{\boldsymbol{\Gamma}}\,\Gamma) \to \mathsf{Tm}_{\boldsymbol{\Gamma}}\,\Gamma\,A$$

In short, an algebra is inductive if every displayed algebra over it has a section. Fortunately, we also know that induction and initiality are equivalent.

Theorem 1. *An object* $\Gamma : \mathsf{Con}_{\boldsymbol{\Gamma}}$ *in an flcwf* $\boldsymbol{\Gamma}$ *supports induction if and only if it is initial. Moreover, induction and initiality are both mere properties.*

Proof. First, we show that induction implies initiality. We assume $\Gamma : \mathsf{Con}$, $\mathsf{ind} : \mathsf{Inductive}\,\Gamma$ and $\Delta : \mathsf{Con}$. We aim to show that there is a unique inhabitant of $\mathsf{Sub}\,\Gamma\,\Delta$. We have $\mathsf{ind}\,(\mathsf{K}\,\Delta) : \mathsf{Tm}\,\Gamma\,(\mathsf{K}\,\Delta)$, hence $\mathsf{ind}\,(\mathsf{K}\,\Delta) : \mathsf{Sub}\,\Gamma\,\Delta$. We only need to show that this is unique. Assume $\delta : \mathsf{Sub}\,\Gamma\,\Delta$. Now, $\mathsf{ind}\,(\mathsf{Id}\,\delta\,(\mathsf{ind}\,(\mathsf{K}\,\Delta))) : \mathsf{Tm}\,\Gamma\,(\mathsf{Id}\,\delta\,(\mathsf{ind}\,(\mathsf{K}\,\Delta)))$, and it follows by equality reflection that $\delta \equiv \mathsf{ind}\,(\mathsf{K}\,\Delta)$.

Second, the other direction. We assume that Γ is initial, and also $A : \mathsf{Ty}\,\Gamma$, and aim to inhabit $\mathsf{Tm}\,\Gamma\,A$. By initiality we get a unique $\sigma : \mathsf{Sub}\,\Gamma\,(\Gamma \triangleright A)$. Now, $\mathsf{q}[\sigma] : \mathsf{Tm}\,\Gamma\,(A[\mathsf{p} \circ \sigma])$, but since $\mathsf{p} \circ \sigma : \mathsf{Sub}\,\Gamma\,\Gamma$, it must be equal to id by the initiality of Γ. Hence, $\mathsf{q}[\sigma] : \mathsf{Tm}\,\Gamma\,A$.

Lastly: it is well-known that initiality is a mere property, so let's show the same for induction. We assume $\mathsf{ind}, \mathsf{ind}'$: $\mathsf{Inductive}\,\Gamma$ and A : $\mathsf{Ty}\,\Gamma$. We have $\mathsf{reflect}\,(\mathsf{ind}\,(\mathsf{Id}\,(\mathsf{ind}\,A)\,(\mathsf{ind}'\,A)))$: $\mathsf{ind}\,A \equiv \mathsf{ind}'\,A$. Since A is arbitrary, by function extensionality we also have $\mathsf{ind} \equiv \mathsf{ind}'$. □

Note that the above proof does not rely on Σ-types in the flcwf, so why do we include them in the semantics? One reason is the prior result by Clairmabault and Dybjer [CD14] that a slightly different formulation of flcwfs is biequivalent to finitely complete categories. More concretely, in ibid. there is a 2-category of cwfs with Σ, Id and "democracy", the last of which is equivalent to the weak formulation of constant families. Then, it is shown that this 2-category is biequivalent to the 2-category of finitely complete categories. Thus, including Σ is a good deal, as this allows us to connect our semantics back to finitely complete categories, which are more commonplace in categorical settings.

We recover finite limits in an flcwf as follows. The product of Γ and Δ is given by $\Gamma \triangleright \mathsf{K}\,\Delta$, and we get projection and pairing from context comprehension. The equalizer of σ, δ : $\mathsf{Sub}\,\Gamma\,\Delta$ is given by $\Gamma \triangleright \mathsf{Id}\,\sigma\,\delta$, which is well-typed because morphisms can be viewed as terms, e.g. σ : $\mathsf{Tm}\,\Gamma\,(\mathsf{K}\,\Delta)$. The unique morphism out of the equalizer is p : $\mathsf{Sub}\,(\Gamma \triangleright \mathsf{Id}\,\sigma\,\delta)\,\Gamma$.

Our flcwf is not exactly the same as in [CD14] because our constant families are strict. But we certainly don't lose anything by having stricter semantics, since the weak version can be trivially recovered.

In the following we present some concepts and results in flcwfs.

Definition 43 (Type categories, c.f. [CD14, Section 2.2])**.** For each Γ : Con, there is a category whose objects are types A : $\mathsf{Ty}\,\Gamma$, and morphisms from A to B are terms t : $\mathsf{Tm}\,(\Gamma \triangleright A)\,(B[\mathsf{p}])$. Identity morphisms are given by q : $\mathsf{Tm}\,(\Gamma \triangleright A)\,(A[\mathsf{p}])$, and composition $t \circ u$ by $t[\mathsf{p}, u]$. The assignment of type categories to contexts extends to a split indexed category. For each σ : $\mathsf{Sub}\,\Gamma\,\Delta$, there is a functor from $\mathsf{Ty}\,\Delta$ to $\mathsf{Ty}\,\Gamma$, which sends A to $A[\sigma]$ and t : $\mathsf{Tm}\,(\Gamma \triangleright A)\,(B[\mathsf{p}])$ to $t[\sigma \circ \mathsf{p}, \mathsf{q}]$.

Notation 15.

- In any cwf, we use σ : $\Gamma \simeq \Delta$ to indicate that σ : $\mathsf{Sub}\,\Gamma\,\Delta$ is an isomorphism with inverse σ^{-1}.

- A *type isomorphism*, written as t : $A \simeq B$ is an isomorphism in a type category, with inverse as t^{-1}.

Theorem 2 (Equivalence of types and slices, c.f. [CD14, Section 2.2])**.** *Assume that we work in an flcwf* Γ*. For each* $\Gamma : \mathsf{Con}$*, the type category* $\mathsf{Ty}\,\Gamma$ *is equivalent to the slice category* Γ/Γ*.*

Remark. In the flcwf of sets where types are $A \to \mathsf{Set}$ families, the above theorem yields the equivalence of $A \to \mathsf{Set}$ and $(B : \mathsf{Set}) \times (B \to A)$. This is sometimes called the "family-fibration" equivalence. It is also a notable motivating example for univalence in type theory: it is not an isomorphism of sets, but only an equivalence up to isomorphism of sets. So this is an example for an equivalence which quite naturally arises even if we only care about sets, but one which is not covered by set-level univalence, and actually requires univalence for groupoids, if we want to prove it as a propositional equality.

4.2.4   The Cwf of Finite Limit Cwfs

The next task is to define the cwf part of **M**. We already know that objects are flcwfs.

Category

A **morphism** $\boldsymbol{\sigma}$: **Sub** $\boldsymbol{\Gamma}$ $\boldsymbol{\Delta}$ is in an algebra homomorphism, viewing flcwfs as algebraic structures. Hence, $\boldsymbol{\sigma}$ includes a functor between underlying categories, but it also maps types to types and terms to terms, and *strictly preserves all structure.*

Notation 16. We may implicitly project out the underlying maps from $\boldsymbol{\sigma}$. Hence, we have the following four maps:

$$\boldsymbol{\sigma} : \mathsf{Con}_{\boldsymbol{\Gamma}} \to \mathsf{Con}_{\boldsymbol{\Delta}}$$
$$\boldsymbol{\sigma} : \mathsf{Sub}_{\boldsymbol{\Gamma}}\,\Gamma\,\Delta \to \mathsf{Sub}_{\boldsymbol{\Delta}}\,(\boldsymbol{\sigma}\,\Gamma)\,(\boldsymbol{\sigma}\,\Delta)$$
$$\boldsymbol{\sigma} : \mathsf{Ty}_{\boldsymbol{\Gamma}}\,\Gamma \to \mathsf{Ty}_{\boldsymbol{\Delta}}\,(\boldsymbol{\sigma}\,\Gamma)$$
$$\boldsymbol{\sigma} : \mathsf{Tm}_{\boldsymbol{\Gamma}}\,\Gamma\,A \to \mathsf{Tm}_{\boldsymbol{\Delta}}\,(\boldsymbol{\sigma}\,\Gamma)\,(\boldsymbol{\sigma}\,A)$$

We list some of the preservation equations as examples of usage:

$$\boldsymbol{\sigma} \bullet \equiv \bullet$$
$$\boldsymbol{\sigma} (\Gamma \triangleright A) \equiv \boldsymbol{\sigma} \Gamma \triangleright \boldsymbol{\sigma} A$$
$$\boldsymbol{\sigma} (A[\sigma]) \equiv (\boldsymbol{\sigma} A)[\boldsymbol{\sigma} \sigma]$$
$$\boldsymbol{\sigma} (t[\sigma]) \equiv (\boldsymbol{\sigma} t)[\boldsymbol{\sigma} \sigma]$$
$$\boldsymbol{\sigma} (\Sigma A B) \equiv \Sigma (\boldsymbol{\sigma} A) (\boldsymbol{\sigma} B)$$
$$\boldsymbol{\sigma} (\mathsf{proj}_1 t) \equiv \mathsf{proj}_1 (\boldsymbol{\sigma} t)$$

Above, we could have also included subscripts indicating the $\boldsymbol{\Gamma}$ or $\boldsymbol{\Delta}$ flcwf, as in $\boldsymbol{\sigma} \bullet_{\boldsymbol{\Gamma}} \equiv \bullet_{\boldsymbol{\Delta}}$; but these are quite easily inferable, so we omit them.

Identity morphisms and **composition** are defined in the evident way using identity functions and function composition in underlying maps, and they satisfy the category laws.

The terminal object $\bullet$: **Con** is given by having $\mathsf{Con}_\bullet :\equiv \top$, $\mathsf{Sub}_\bullet \Gamma \Delta :\equiv \top$, $\mathsf{Ty}_\bullet \Gamma :\equiv \top$ and $\mathsf{Tm}_\bullet \Gamma A :\equiv \top$, and all structure and equations are defined trivially.

Family Structure

A type $\boldsymbol{A}$: **Ty** $\boldsymbol{\Gamma}$ is a displayed flcwf over $\boldsymbol{\Gamma}$. As we have seen before, displayed algebras can be computed as logical predicate interpretations of algebraic signatures. Every $\boldsymbol{A}$ component lies over the corresponding $\boldsymbol{\Gamma}$ component. Also note that a displayed flcwf includes a displayed category, for which some results have been worked out in [AL19].

Notation 17. In situations where we need to refer to both "base" and displayed things, we give <u>underlined</u> names to contexts, substitutions, types and terms in a base flcwf. For example, we may have $\underline{\Gamma}$: $\mathsf{Con}_{\boldsymbol{\Gamma}}$ living in $\boldsymbol{\Gamma}$: **Con**, and Γ : $\mathsf{Con}_{\boldsymbol{A}} \underline{\Gamma}$ living in a displayed flcwf over $\boldsymbol{\Gamma}$. We only use underlining on 2LTT variable names, and overload flcwf component names for displayed counterparts. For example, a Con component is named the same in a base flcwf and a displayed one.

Concretely, a displayed flcwf $\boldsymbol{A}$ over $\boldsymbol{\Gamma}$ has the following underlying sets, which

we call displayed contexts, substitutions, types and terms respectively.

$$\mathsf{Con}_{A} : \mathsf{Con}_{\Gamma} \to \mathsf{Set}$$

$$\mathsf{Sub}_{A} : \mathsf{Con}_{A}\,\underline{\Gamma} \to \mathsf{Con}_{A}\,\underline{\Delta} \to \mathsf{Sub}_{\Gamma}\,\underline{\Gamma}\,\underline{\Delta} \to \mathsf{Set}$$

$$\mathsf{Ty}_{A} \ : \mathsf{Con}_{A}\,\underline{\Gamma} \to \mathsf{Ty}_{\Gamma}\,\underline{\Gamma} \to \mathsf{Set}$$

$$\mathsf{Tm}_{A} \ : (\Gamma : \mathsf{Con}_{A}\,\underline{\Gamma}) \to \mathsf{Ty}_{A}\,\Gamma\,\underline{A} \to \mathsf{Tm}_{\Gamma}\,\underline{\Gamma}\,\underline{A} \to \mathsf{Set}$$

We list select components of $\boldsymbol{A}$ below; note how every $\boldsymbol{A}$ operation lies over the corresponding $\boldsymbol{\Gamma}$ operation. In our notation with implicit arguments, equations in $\boldsymbol{A}$ can be written the same way as in $\boldsymbol{\Gamma}$, but of course there is extra indexing involved, and the displayed equations are well-typed because of their counterparts in the base.

$$\mathsf{id}_{A} \qquad : \mathsf{Sub}\,\Gamma\,\Gamma\,\underline{\mathsf{id}_{\Gamma}}$$

$$- \circ_{A} - \ : \mathsf{Sub}\,\Delta\,\Xi\,\underline{\sigma} \to \mathsf{Sub}\,\Gamma\,\Delta\,\underline{\delta} \to \mathsf{Sub}\,\Gamma\,\Xi\,(\underline{\sigma} \circ_{\Gamma} \underline{\delta})$$

$$\mathsf{idl}_{A} \qquad : \mathsf{id}_{A} \circ_{A} \sigma \equiv \sigma$$

$$\mathsf{idr}_{A} \qquad : \sigma \circ_{A} \mathsf{id}_{A} \equiv \sigma$$

$$\bullet_{A} \qquad : \mathsf{Con}_{A}\,\bullet_{\Gamma}$$

$$- \triangleright_{A} - \ : (\Gamma : \mathsf{Con}_{A}\,\underline{\Gamma}) \to \mathsf{Ty}_{A}\,\Gamma\,\underline{A} \to \mathsf{Con}_{A}\,\Gamma\,(\underline{\Gamma} \triangleright_{\Gamma} \underline{A})$$

$$-[-]_{A} \ : \mathsf{Ty}_{A}\,\Delta\,\underline{A} \to \mathsf{Sub}_{A}\,\Gamma\,\Delta\,\underline{\sigma} \to \mathsf{Ty}_{A}\,\Gamma\,(\underline{A}[\underline{\sigma}]_{\Gamma})$$

$$-[-]_{A} \ : \mathsf{Tm}_{A}\,\Delta\,A\,\underline{t} \to (\sigma : \mathsf{Sub}_{A}\,\Gamma\,\Delta\,\underline{\sigma}) \to \mathsf{Tm}_{A}\,\Gamma\,(A[\sigma]_{A})\,(\underline{t}[\underline{\sigma}]_{\Gamma})$$

$$\mathsf{Id}_{A} \qquad : \mathsf{Tm}_{A}\,\Gamma\,A\,\underline{t} \to \mathsf{Tm}_{A}\,\Gamma\,A\,\underline{u} \to \mathsf{Ty}_{A}\,\Gamma\,(\mathsf{Id}_{\Gamma}\,\underline{t}\,\underline{u})$$

$$\mathsf{K}_{A} \qquad : \mathsf{Con}_{A}\,\underline{\Delta} \to \{\Gamma : \mathsf{Con}_{A}\,\underline{\Gamma}\} \to \mathsf{Ty}_{A}\,\Gamma\,(\mathsf{K}_{\Gamma}\,\underline{\Delta})$$

$$\Sigma_{A} \qquad : (A : \mathsf{Ty}_{A}\,\Gamma\,\underline{A}) \to \mathsf{Ty}_{A}\,(\Gamma \triangleright_{A} A)\,\underline{B} \to \mathsf{Ty}_{A}\,\Gamma\,(\Sigma_{\Gamma}\,\underline{A}\,\underline{B})$$

In the following we will often omit $_{\Gamma}$ and $_{A}$ subscripts on components; for example, in the type $\mathsf{Con}_{A}\,\bullet$, the $\bullet$ is clearly a base component in $\boldsymbol{\Gamma}$.

A substituted type $\boldsymbol{A[\sigma]} : \mathbf{Ty}\,\boldsymbol{\Gamma}$ is defined as follows, for $\boldsymbol{A} : \mathbf{Ty}\,\boldsymbol{\Delta}$ and $\boldsymbol{\sigma} : \mathbf{Sub}\,\boldsymbol{\Gamma}\,\boldsymbol{\Delta}$. We simply compose underlying functions in $\boldsymbol{\sigma}$ with the underlying predicates in $\boldsymbol{A}$:

$$\mathsf{Con}_{A[\sigma]}\,\Gamma \qquad :\equiv \mathsf{Con}_{A}\,(\boldsymbol{\sigma}\,\Gamma)$$

$$\mathsf{Sub}_{A[\sigma]}\,\Gamma\,\Delta\,\underline{\sigma} :\equiv \mathsf{Sub}_{A}\,\Gamma\,\Delta\,(\boldsymbol{\sigma}\,\underline{\sigma})$$

$$\mathsf{Ty}_{A[\sigma]}\,\Gamma\,\underline{A} \qquad :\equiv \mathsf{Ty}_{A}\,\Gamma\,(\boldsymbol{\sigma}\,\underline{A})$$

$$\mathsf{Tm}_{A[\sigma]}\,\Gamma\,A\,\underline{t} \ :\equiv \mathsf{Tm}_{A}\,\Gamma\,A\,(\boldsymbol{\sigma}\,\underline{t})$$

It should be clear that $\boldsymbol{A}[\boldsymbol{\sigma}]$ thus defined still supports all displayed flcwf struc-
ture. For example, the displayed contexts in $\boldsymbol{A}[\boldsymbol{\sigma}]$ are elements of $\mathsf{Con}_{\boldsymbol{A}}\,(\boldsymbol{\sigma}\,\underline{\Gamma})$,
but since $\boldsymbol{\sigma}$ preserves all $\boldsymbol{\Gamma}$-structure, we can also recover all displayed structure.
For example, if $\underline{\Gamma}$ is $\underline{\bullet}$, we have $\boldsymbol{\sigma}\,\underline{\bullet}\equiv\underline{\bullet}$, and we can reuse $\bullet_{\boldsymbol{A}}:\mathsf{Con}_{\boldsymbol{A}}\,\underline{\bullet}$ to define the
displayed empty context in $\boldsymbol{A}[\boldsymbol{\sigma}]$, and we can proceed analogously for all other
structure in $\boldsymbol{A}[\boldsymbol{\sigma}]$.

Additionally, type substitution is functorial, i.e. $\boldsymbol{A}[\mathsf{id}]\equiv\boldsymbol{A}$ and $\boldsymbol{A}[\boldsymbol{\sigma}\circ\boldsymbol{\delta}]\equiv$
$\boldsymbol{A}[\boldsymbol{\sigma}][\boldsymbol{\delta}]$. This holds because the underlying set families are defined by function
composition.

Remark. Types could be equivalently defined as slices in the category of flcws,
and type substitution could be given as pullback, but in that case we would run
into the well-known strictness issue, that type substitution is functorial only up
to isomorphism [**?**]. This is not a critical issue, as there are standard solutions
for recovering strict substitutions from weak ones [**?**]. But if we ever need to
look inside the definitions in the model, using displayed algebras yields a lot less
encoding overhead than strictifying pullbacks.

A term $\boldsymbol{t}:\mathbf{Tm}\,\Gamma\,\boldsymbol{A}$ is a displayed flcwf section, which again strictly preserves
all structure. We use the same notation for the action of $\boldsymbol{t}$ that we use for **Sub**.
We have the following underlying maps:

$$\boldsymbol{t}:(\underline{\Gamma}:\mathsf{Con}_{\boldsymbol{\Gamma}})\to\mathsf{Con}_{\boldsymbol{A}}\,\underline{\Gamma}$$
$$\boldsymbol{t}:(\underline{\sigma}:\mathsf{Sub}_{\boldsymbol{\Gamma}}\,\Gamma\,\Delta)\to\mathsf{Sub}_{\boldsymbol{A}}\,(\boldsymbol{t}\,\Gamma)\,(\boldsymbol{t}\,\Delta)\,\underline{\sigma}$$
$$\boldsymbol{t}:(\underline{A}:\mathsf{Ty}_{\boldsymbol{\Gamma}}\,\Gamma)\to\mathsf{Ty}_{\boldsymbol{A}}\,(\boldsymbol{t}\,\Gamma)\,\underline{A}$$
$$\boldsymbol{t}:(\underline{t}:\mathsf{Tm}_{\boldsymbol{\Gamma}}\,\Gamma\,A)\to\mathsf{Tm}_{\boldsymbol{A}}\,(\boldsymbol{t}\,\Gamma)\,(\boldsymbol{t}\,A)\,\underline{t}$$

A substituted term $\boldsymbol{t}[\boldsymbol{\sigma}]$ for $\boldsymbol{t}:\mathbf{Tm}\,\Delta\,\boldsymbol{A}$ and $\boldsymbol{\sigma}:\mathbf{Sub}\,\Gamma\,\Delta$ is again given by
component-wise function composition.

An extended context $\boldsymbol{\Gamma}\triangleright\boldsymbol{A}$ is the *total flcwf* of $\boldsymbol{A}$. This is defined by combining
corresponding underlying sets with Σ-types:

$$\mathsf{Con}_{\boldsymbol{\Gamma}\triangleright\boldsymbol{A}}:\equiv(\underline{\Gamma}:\mathsf{Con}_{\boldsymbol{\Gamma}})\times\mathsf{Con}_{\boldsymbol{A}}\,\underline{\Gamma}$$
$$\mathsf{Sub}_{\boldsymbol{\Gamma}\triangleright\boldsymbol{A}}\,(\underline{\Gamma},\,\Gamma)\,(\underline{\Delta},\,\Delta):\equiv(\underline{\sigma}:\mathsf{Sub}_{\boldsymbol{\Gamma}}\,\underline{\Gamma}\,\underline{\Delta})\times\mathsf{Sub}_{\boldsymbol{A}}\,\Gamma\,\Delta\,\underline{\sigma}$$
$$\mathsf{Ty}_{\boldsymbol{\Gamma}\triangleright\boldsymbol{A}}\,(\underline{\Gamma},\,\Gamma):\equiv(\underline{A}:\mathsf{Ty}_{\boldsymbol{\Gamma}}\,\underline{\Gamma})\times\mathsf{Ty}_{\boldsymbol{A}}\,\Gamma\,\underline{A}$$
$$\mathsf{Tm}_{\boldsymbol{\Gamma}\triangleright\boldsymbol{A}}\,(\underline{\Gamma},\,\Gamma)\,(\underline{A},\,A):\equiv(\underline{t}:\mathsf{Tm}_{\boldsymbol{\Gamma}}\,\underline{\Gamma}\,\underline{A})\times\mathsf{Tm}_{\boldsymbol{A}}\,\Gamma\,A\,\underline{t}$$

All structure is defined pointwise, using $\boldsymbol{\Gamma}$-structure for first projections and $\boldsymbol{A}$-

structure for second projections. $\mathbf{\Gamma} \triangleright \boldsymbol{A}$ may be viewed as a dependent generalization of direct products of flcwfs.

Comprehension structure follows from the above definition: $\mathbf{p}$ is componentwise first projection, $\mathbf{q}$ is second projection and substitution extension $-\,,-$ is pairing.

With this, we have a cwf of flcws. *Remark:* flcwf itself is algebraic, and has a finitary QII signature. Hence, if we succeed building semantics for finitary QII signatures, we get "for free" an flcwf of flcwfs. Of course, we cannot rely on this when we're in the process of defining the **M** model in the first place. Checking that the **M** model indeed works, is the somewhat tedious task that we have to perform *once*, in order to get semantics for any other finitary QII theory.

4.2.5 Type Formers

Strict Constant Families

This was not included in the ToS specification, but it's quite useful, so we shall define it. $\mathbf{K}\,\Delta : \mathbf{Ty}\,\Gamma$ is defined by ignoring $\mathbf{\Gamma}$ inhabitants in all underlying sets:

$$
\begin{aligned}
\mathsf{Con}_{\mathbf{K}\,\Delta}\,\underline{\Gamma} &:\equiv \mathsf{Con}_{\Delta} \\
\mathsf{Sub}_{\mathbf{K}\,\Delta}\,\Gamma\,\Delta\,\underline{\sigma} &:\equiv \mathsf{Sub}_{\Delta}\,\Gamma\,\Delta \\
\mathsf{Ty}_{\mathbf{K}\,\Delta}\,\Gamma\,\underline{A} &:\equiv \mathsf{Ty}_{\Delta}\,\Gamma \\
\mathsf{Tm}_{\mathbf{K}\,\Delta}\,\Gamma\,A\,\underline{t} &:\equiv \mathsf{Tm}_{\Delta}\,\Gamma\,A
\end{aligned}
$$

All structure is inherited from $\mathbf{\Delta}$. There is also a type substitution rule, expressing that for $\boldsymbol{\sigma} : \mathbf{Sub}\,\Gamma\,\Xi$, we have $(\mathbf{K}\,\{\Xi\}\,\Delta)[\sigma] \equiv \mathbf{K}\,\{\Gamma\}\,\Delta$. This follows immediately from the above definition and the definition of type substitution, since the base inhabitants are ignored the same way on both sides of the equation. We also need to show $\mathbf{Tm}\,\Gamma\,(\mathbf{K}\,\Delta) \equiv \mathbf{Sub}\,\Gamma\,\Delta$. This again follows directly from the $\mathbf{K}$ definition. From $\mathbf{K}$, we get

- The unit type, defined as $\mathbf{K}\,\bullet : \mathbf{Ty}\,\Gamma$.

- Direct products of $\mathbf{\Gamma}$ and $\mathbf{\Delta}$, defined as $\mathbf{\Gamma} \triangleright \mathbf{K}\,\Delta$.

- The ability to define closed type formers as elements of $\mathbf{Con}$.

Universe

Similarly to what we did in Definition 32, we define **U** as a context, and use **K** later to get the universe as a type. **U** : **Con** is defined to be the flcwf where objects are inner types, and morphisms are outer functions between them:

$$
\begin{aligned}
\mathsf{Con}_{\mathsf{U}} &:\equiv \mathsf{Ty}_0 \\
\mathsf{Sub}_{\mathsf{U}}\,\Gamma\,\Delta &:\equiv \mathsf{Tm}_0\,\Gamma \to \mathsf{Tm}_0\,\Delta \\
\mathsf{Ty}_{\mathsf{U}}\,\Gamma &:\equiv \mathsf{Tm}_0\,\Gamma \to \mathsf{Ty}_0 \\
\mathsf{Tm}_{\mathsf{U}}\,\Gamma\,A &:\equiv (\gamma : \mathsf{Tm}_0\,\Gamma) \to \mathsf{Tm}_0\,(A\,\gamma)
\end{aligned}
$$

Substitution for types and terms is defined by function composition. The empty context is defined as the inner unit type $\top_0$, and context extension $\Gamma \triangleright_{\mathsf{U}} A$ is defined as $(\gamma : \Gamma) \times A\,\gamma$ using inner Σ. We can also define Σ_{U} and Id_{U} using inner Σ and identity.

For constant families, we don't need any additional assumption in the inner theory, since it can be defined as $\mathsf{K}_{\mathsf{U}}\,\{\Gamma\}\,\Delta :\equiv \Delta$, and $\mathsf{Sub}_{\mathsf{U}}\,\Gamma\,\Delta \equiv \mathsf{Tm}_{\mathsf{U}}\,\Gamma\,(\mathsf{K}_{\mathsf{U}}\,\Delta)$ follows immediately.

For $\boldsymbol{a}$: **Sub** Γ **U**, we have to define **El** $\boldsymbol{a}$: **Ty** Γ. This is given as the *displayed flcwf of elements* of $\boldsymbol{a}$.

Background: from any functor $F : \mathbb{C} \to \mathbf{Set}$ we can construct the category of elements $\int F$, where objects are in $(i : |\mathbb{C}|) \times F\,i$ and morphisms between (i,x) and (j,y) are in $(f : \mathbb{C}(i,j)) \times (F\,f\,x \equiv y)$. If we take the second projections of components in $\int F$, we get the displayed category of elements, which lies over $\mathbb{C}$. We may also call this a *discrete displayed category*, in analogy to discrete categories, whose objects are elements of sets.

We extend this to flcwfs in the definition of **El** $\boldsymbol{a}$. With this definition, $\Gamma \triangleright \mathbf{El}\,\boldsymbol{a}$ will yield the flcwf of elements of $\boldsymbol{a}$.

$$
\begin{aligned}
\mathsf{Con}_{\mathsf{El}\,a}\,\underline{\Gamma} &:\equiv \mathsf{Tm}_0\,(\boldsymbol{a}\,\underline{\Gamma}) \\
\mathsf{Sub}_{\mathsf{El}\,a}\,\Gamma\,\Delta\,\underline{\sigma} &:\equiv \boldsymbol{a}\,\underline{\sigma}\,\Gamma \equiv \Delta \\
\mathsf{Ty}_{\mathsf{El}\,a}\,\Gamma\,\underline{A} &:\equiv \mathsf{Tm}_0\,(\boldsymbol{a}\,\underline{A}\,\Gamma) \\
\mathsf{Tm}_{\mathsf{El}\,a}\,\Gamma\,A\,\underline{t} &:\equiv \boldsymbol{a}\,\underline{t}\,\Gamma \equiv A
\end{aligned}
$$

Let's check that we have all other structure as well.

- For contexts and types, the task is to exhibit elements of $\boldsymbol{a}$ lying over specific base contexts and types.

- For terms and substitutions, the task is to exhibit equations which specify the action of $\boldsymbol{a}$.

- Equations between terms and substitutions are trivial because of UIP (we need to show equations between equality proofs).

We summarize below the additional structure on top of the displayed category part of $\mathsf{El}\,\boldsymbol{a}$.

- For $\bullet_{\mathsf{El}\,\boldsymbol{a}} : \mathsf{Con}_{\mathsf{El}\,\boldsymbol{a}}\,\underline{\bullet}$, the type can be simplified along the definition of $\mathsf{Con}_{\mathsf{El}\,\boldsymbol{a}}$ and structure-preservation by $\boldsymbol{a}$ to $\mathsf{Tm}_0\,\top_0$. Hence, $\bullet_{\mathsf{El}\,\boldsymbol{a}} :\equiv \mathsf{tt}_0$ is the unique definition. For $\epsilon : \mathsf{Sub}_{\mathsf{El}\,\boldsymbol{a}}\,\Gamma\,\bullet_{\mathsf{El}\,\boldsymbol{a}}\,\underline{\epsilon}$, we have to show $\boldsymbol{a}\,\underline{\epsilon}\,\Gamma \equiv \mathsf{tt}_0$, which holds by the uniqueness of tt_0.

- For $\Gamma \rhd_{\mathsf{El}\,\boldsymbol{a}} A : \mathsf{Con}_{\mathsf{El}\,\boldsymbol{a}}\,(\underline{\Gamma} \rhd \underline{A})$, the target type unfolds to $\mathsf{Tm}_0\,(\boldsymbol{a}\,(\underline{\Gamma} \rhd \underline{A}))$, which in turn simplifies to $\mathsf{Tm}_0\,((\gamma : \boldsymbol{a}\,\underline{\Gamma}) \times \boldsymbol{a}\,\underline{A}\,\gamma)$. Since $\Gamma : \mathsf{Tm}_0\,(\boldsymbol{a}\,\underline{\Gamma})$ and $A : \mathsf{Tm}_0\,(\boldsymbol{a}\,\underline{A}\,\Gamma)$, we define $\Gamma \rhd_{\mathsf{El}\,\boldsymbol{a}} A$ as $(\Gamma,\,A)$.

- For comprehension, we have to show the following, after simplifying types:

$$
\begin{aligned}
\mathsf{p} &\quad : \boldsymbol{a}\,\underline{\mathsf{p}}\,(\Gamma,\,A) \equiv \Gamma \\
\mathsf{q} &\quad : \boldsymbol{a}\,\underline{\mathsf{q}}\,(\Gamma,\,A) \equiv A \\
(\sigma,\,t) &: \boldsymbol{a}\,(\underline{\sigma},\,\underline{t})\,\Gamma \equiv (\Delta,\,A)
\end{aligned}
$$

For p and q, equations follow from preservation by $\boldsymbol{a}$. For pairing, the goal further simplifies to $(\boldsymbol{a}\,\underline{\sigma}\,\Gamma,\,\boldsymbol{a}\,\underline{t}\,\Gamma) \equiv (\Delta,\,A)$. Then, the first and second components are equal by the σ and t hypotheses.

- Assuming $A : \mathsf{Ty}_{\mathsf{El}\,\boldsymbol{a}}\,\Delta\,\underline{A}$ and $\sigma : \mathsf{Sub}_{\mathsf{El}\,\boldsymbol{a}}\,\Gamma\,\Delta\,\underline{\sigma}$, we aim to define $A[\sigma]_{\mathsf{El}\,\boldsymbol{a}} : \mathsf{Ty}_{\mathsf{El}\,\boldsymbol{a}}\,\Gamma\,(\underline{A}[\underline{\sigma}])$. Simplifying types, $A : \mathsf{Tm}_0\,(\boldsymbol{a}\,\underline{A}\,\Delta)$, $\sigma : \boldsymbol{a}\,\underline{\sigma}\,\Gamma \equiv \Delta$ and the target type is $\mathsf{Tm}_0\,(\boldsymbol{a}\,(\underline{A}[\underline{\sigma}])\,\Gamma)$, which is the same as $\mathsf{Tm}_0\,(\boldsymbol{a}\,\underline{A}\,(\boldsymbol{a}\,\underline{\sigma}\,\Gamma))$, by the preservation of $-[-]$ by $\boldsymbol{a}$. Hence, by the σ assumption, the target type is $\mathsf{Tm}_0\,(\boldsymbol{a}\,\underline{A}\,\Delta)$, so we give the following definition:

$$
A[\sigma]_{\mathsf{El}\,\boldsymbol{a}} :\equiv A
$$

This is clearly functorial; moreover, substitution rules for the other type formers hold trivially.

- Term substitution is given by transitivity of equality.

- For $\mathsf{Id}_{\mathsf{El}\,a}\,t\,u : \mathsf{Ty}_{\mathsf{El}\,a}\,\Gamma\,(\mathsf{Id}\,\underline{t}\,\underline{u})$, the goal type is $\mathsf{Tm}_0\,(\boldsymbol{a}\,(\mathsf{Id}\,\underline{t}\,\underline{u})\,\Gamma)$, hence $\mathsf{Tm}_0\,(\boldsymbol{a}\,\underline{t}\,\Gamma = \boldsymbol{a}\,\underline{u}\,\Gamma)$. This holds by $t : \boldsymbol{a}\,\underline{t}\,\Gamma \equiv A$ and $u : \boldsymbol{a}\,\underline{t}\,\Gamma \equiv A$. Reflexivity and equality reflection are trivial by UIP.

- For $A : \mathsf{Ty}_{\mathsf{El}\,a}\,\Gamma\,\underline{A}$ and $B : \mathsf{Ty}_{\mathsf{El}\,a}\,(\Gamma \triangleright A)\,\underline{B}$, we aim to define $\Sigma_{\mathsf{El}\,a}\,A\,B : \mathsf{Ty}_{\mathsf{El}\,a}\,\Gamma\,(\Sigma\,\underline{A}\,\underline{B})$, hence

$$\Sigma_{\mathsf{El}\,a}\,A\,B : \mathsf{Tm}_0\,(\boldsymbol{a}\,(\Sigma\,\underline{A}\,\underline{B})\,\Gamma)$$
$$\Sigma_{\mathsf{El}\,a}\,A\,B : \mathsf{Tm}_0\,((A : \boldsymbol{a}\,\underline{A}\,\Gamma) \times \boldsymbol{a}\,\underline{B}\,(\Gamma,\,A))$$
$$\Sigma_{\mathsf{El}\,a}\,A\,B :\equiv (A,\,B)$$

  Projections and pairing proceed analogously to what we did for comprehension.

- For $\mathsf{K}_{\mathsf{El}\,a}\,\Delta : \mathsf{Ty}_{\mathsf{El}\,a}\,\Gamma\,(\mathsf{K}\,\underline{\Delta})$, the target type simplifies to $\mathsf{Tm}_0\,(\boldsymbol{a}\,\underline{\Delta})$, hence we have $\mathsf{K}_{\mathsf{El}\,a}\,\Delta :\equiv \Delta$. For the specifying sort equation of K, we have to show

$$\mathsf{Sub}_{\mathsf{El}\,a}\,\Gamma\,\Delta\,\underline{\sigma} \equiv \mathsf{Tm}_{\mathsf{El}\,a}\,\Gamma\,(\mathsf{K}_{\mathsf{El}\,a}\,\Delta)\,\underline{\sigma}$$

  where $\underline{\sigma} : \mathsf{Sub}\,\underline{\Gamma}\,\underline{\Delta}$ but at the same time $\underline{\sigma} : \mathsf{Tm}\,\underline{\Gamma}\,(\mathsf{K}\,\underline{\Delta})$ because of the K sort equation in the base. Fortunately, both sides simplify to $\boldsymbol{a}\,\underline{\sigma}\,\Gamma \equiv \Delta$.

We still have to check $(\mathbf{El}\,\boldsymbol{a})[\boldsymbol{\sigma}] \equiv \mathbf{El}\,(\boldsymbol{a} \circ \boldsymbol{\sigma})$, the naturality rule for $\mathbf{El}$. We only have to check equality of underlying sets, Con and Ty formers, since terms and substitutions are equal by UIP. For underlying sets, both sides compute to the following:

$$\mathsf{Con}\,\underline{\Gamma} \quad :\equiv \mathsf{Tm}_0\,(\boldsymbol{a}\,(\boldsymbol{\sigma}\,\underline{\Gamma}))$$
$$\mathsf{Sub}\,\Gamma\,\Delta\,\underline{\sigma} :\equiv \boldsymbol{a}\,(\boldsymbol{\sigma}\,\underline{\sigma})\,\Gamma \equiv \Delta$$
$$\mathsf{Ty}\,\Gamma\,\underline{A} \quad :\equiv \mathsf{Tm}_0\,(\boldsymbol{a}\,(\boldsymbol{\sigma}\,\underline{A})\,\Gamma)$$
$$\mathsf{Tm}\,\Gamma\,A\,\underline{t} \quad :\equiv \boldsymbol{a}\,(\boldsymbol{\sigma}\,\underline{t})\,\Gamma \equiv A$$

Since $\boldsymbol{\sigma}$ also strictly preserves all structure, and we simply replace $\boldsymbol{a}$ action by the composite $\boldsymbol{a} \circ \boldsymbol{\sigma}$ action, it is straightforward to check that Con and Ty formers are also the same on both sides.

At this point, we have **U : Con** and **El : Sub Γ U**. Let's rename them to **U$'$** and **El$'$** respectively, and define the usual "open" versions:

$$\mathsf{U} : \mathsf{Ty}\,\Gamma \qquad \mathsf{El} : \mathsf{Tm}\,\Gamma\,\mathsf{U} \to \mathsf{Ty}\,\Gamma$$
$$\mathsf{U} :\equiv \mathsf{K}\,\mathsf{U}' \qquad \mathsf{El}\,a :\equiv \mathsf{El}'\,a$$

Identity

Assuming $t,\,u : \mathbf{Tm}\,\Gamma\,A$, extensional identity **Id $t\,u$** is defined as component-wise equality:

$$\mathsf{Con}_{\mathsf{Id}\,t\,u}\,\underline{\Gamma} \qquad :\equiv t\,\underline{\Gamma} \equiv u\,\underline{\Gamma}$$
$$\mathsf{Sub}_{\mathsf{Id}\,t\,u}\,\Gamma\,\Delta\,\underline{\sigma} :\equiv t\,\underline{\sigma} \equiv u\,\underline{\sigma}$$
$$\mathsf{Ty}_{\mathsf{Id}\,t\,u}\,\Gamma\,\underline{A} \qquad :\equiv t\,\underline{A} \equiv u\,\underline{A}$$
$$\mathsf{Tm}_{\mathsf{Id}\,t\,u}\,\Gamma\,A\,\underline{t} \quad :\equiv t\,\underline{t} \equiv u\,\underline{t}$$

All other structure follows from structure-preservation of t and u. For the simplest example, $\bullet_{\mathsf{Id}\,t\,u} : t\,\underline{\bullet} \equiv u\,\underline{\bullet}$ holds because t and u both preserve $\underline{\bullet}$. The rule $(\mathsf{Id}\,t\,u)[\sigma] \equiv \mathsf{Id}\,(t[\sigma])\,(u[\sigma])$ is straightforward to check: we only have to look at the underlying sets, where e.g. both sides have $\mathsf{Con}\,\underline{\Gamma} \equiv (t\,(\sigma\,\underline{\Gamma}) \equiv u\,(\sigma\,\underline{\Gamma}))$. It's also evident that $\mathbf{Tm}\,\Gamma\,(\mathsf{Id}\,t\,u)$ is equivalent to $t \equiv u$, that is, we have reflexivity and equality reflection.

Inductive Function Type

For $a : \mathbf{Tm}\,\Gamma\,\mathbf{U}$ and $B : \mathbf{Ty}\,(\Gamma \triangleright \mathbf{El}\,a)$, we aim to define $\Pi\,a\,B : \mathbf{Ty}\,\Gamma$. This is dependent product of displayed flcwfs, indexed over a *discrete* domain. Discreteness is critical: since morphisms in **El a** are proof-irrelevant and invertible (because they are equations), we avoid the variance issues that preclude general Π-types in the cwf of categories [Joh02, Secion A1.5].

The direct definition would be to define underlying sets as products, indexed over corresponding components in **El a**:

$$\mathsf{Con}_{\Pi\,a\,B}\,\underline{\Gamma} \qquad :\equiv (\gamma : a\,\underline{\Gamma}) \to \mathsf{Con}_B\,(\underline{\Gamma},\,\gamma)$$
$$\mathsf{Sub}_{\Pi\,a\,B}\,\Gamma\,\Delta\,\underline{\sigma} :\equiv \{\gamma : a\,\underline{\Gamma}\}\{\delta : a\,\underline{\Delta}\}(\sigma : \mathsf{Sub}_{\mathsf{El}\,a}\,\gamma\,\delta\,\underline{\sigma}) \to \mathsf{Sub}_B\,(\Gamma\,\gamma)\,(\Delta\,\delta)\,(\underline{\sigma},\,\sigma)$$
$$\mathsf{Ty}_{\Pi\,a\,B}\,\Gamma\,\underline{A} \qquad :\equiv \{\gamma : a\,\underline{\Gamma}\}(\alpha : a\,\underline{A}\,\gamma) \to \mathsf{Ty}_B\,(\Gamma\,\gamma)\,(\underline{A},\,\alpha)$$
$$\mathsf{Tm}_{\Pi\,a\,B}\,\Gamma\,A\,\underline{t} :\equiv \{\gamma : a\,\underline{\Gamma}\}\{\alpha : a\,\underline{A}\,\gamma\}(t : \mathsf{Tm}_{\mathsf{El}\,a}\,\gamma\,\delta\,\underline{t}) \to \mathsf{Tm}_B\,(\Gamma\,\gamma)\,(A\,\alpha)\,(\underline{t},\,t)$$

But just like in Definitions 4 and 6, we can contract the Sub and Tm definitions, since $\mathsf{Sub}_{\mathsf{El}\,a}\,\gamma\,\delta\,\underline{\sigma} \equiv (\boldsymbol{a}\,\underline{\sigma}\,\gamma \equiv \delta)$ and $\mathsf{Tm}_{\mathsf{El}\,a}\,\gamma\,\alpha\,\underline{t} \equiv (\boldsymbol{a}\,\underline{t}\,\gamma \equiv \alpha)$.

$$
\begin{aligned}
\mathsf{Con}_{\boldsymbol{\Pi}\,\boldsymbol{a}\,\boldsymbol{B}}\,\underline{\Gamma} \quad &:\equiv (\gamma : \boldsymbol{a}\,\underline{\Gamma}) \to \mathsf{Con}_{\boldsymbol{B}}\,(\underline{\Gamma},\,\gamma) \\
\mathsf{Sub}_{\boldsymbol{\Pi}\,\boldsymbol{a}\,\boldsymbol{B}}\,\underline{\Gamma}\,\underline{\Delta}\,\underline{\sigma} &:\equiv (\gamma : \boldsymbol{a}\,\underline{\Gamma}) \to \mathsf{Sub}_{\boldsymbol{B}}\,(\underline{\Gamma}\,\gamma)\,(\underline{\Delta}\,(\boldsymbol{a}\,\underline{\sigma}\,\gamma))\,(\underline{\sigma},\,\mathsf{refl}) \\
\mathsf{Ty}_{\boldsymbol{\Pi}\,\boldsymbol{a}\,\boldsymbol{B}}\,\underline{\Gamma}\,\underline{A} \quad &:\equiv \{\gamma : \boldsymbol{a}\,\underline{\Gamma}\}(\alpha : \boldsymbol{a}\,\underline{A}\,\gamma) \to \mathsf{Ty}_{\boldsymbol{B}}\,(\underline{\Gamma}\,\gamma)\,(\underline{A},\,\alpha) \\
\mathsf{Tm}_{\boldsymbol{\Pi}\,\boldsymbol{a}\,\boldsymbol{B}}\,\underline{\Gamma}\,A\,\underline{t} &:\equiv (\gamma : \boldsymbol{a}\,\underline{\Gamma}) \to \mathsf{Tm}_{\boldsymbol{B}}\,(\underline{\Gamma}\,\gamma)\,(\mathsf{A}\,(\boldsymbol{a}\,\underline{t}\,\gamma))\,(\underline{t},\,\mathsf{refl})
\end{aligned}
$$

With the contracted definition, Sub and Tm are only indexed over displayed objects, but not over displayed morphisms or terms anymore. So it is apparent that we cannot have issues with indexing variance. All structure in $\boldsymbol{\Pi}\,\boldsymbol{a}\,\boldsymbol{B}$ is pointwise inherited from $\boldsymbol{B}$. We list some examples below for definitions.

$$
\begin{aligned}
\bullet_{\boldsymbol{\Pi}\,\boldsymbol{a}\,\boldsymbol{B}}\,\gamma \qquad &:\equiv \bullet_{\boldsymbol{B}} \\
(\underline{\Gamma} \rhd_{\boldsymbol{\Pi}\,\boldsymbol{a}\,\boldsymbol{B}} A)\,(\gamma,\,\alpha) &:\equiv (\underline{\Gamma}\,\gamma \rhd_{\boldsymbol{B}} A\,\alpha) \\
\mathsf{id}_{\boldsymbol{\Pi}\,\boldsymbol{a}\,\boldsymbol{B}}\,\gamma \qquad &:\equiv \mathsf{id}_{\boldsymbol{B}} \\
(\sigma \circ_{\boldsymbol{\Pi}\,\boldsymbol{a}\,\boldsymbol{B}} \delta)\,\gamma \qquad &:\equiv \sigma\,\gamma \circ_{\boldsymbol{B}} \delta\,\gamma \\
A[\sigma]_{\boldsymbol{\Pi}\,\boldsymbol{a}\,\boldsymbol{B}}\,\{\gamma\}\,\alpha \quad &:\equiv (A\,\alpha)[\sigma\,\gamma]_{\boldsymbol{B}} \\
\mathsf{K}_{\boldsymbol{\Pi}\,\boldsymbol{a}\,\boldsymbol{B}}\,\Delta\,\alpha \qquad &:\equiv \mathsf{K}_{\boldsymbol{B}}\,(\Delta\,\alpha)
\end{aligned}
$$

For the specifying isomorphism $(\mathbf{app},\,\mathbf{lam}) : \mathbf{Tm}\,\Gamma\,(\boldsymbol{\Pi}\,\boldsymbol{a}\,\boldsymbol{B}) \simeq \mathbf{Tm}\,(\Gamma \rhd \mathbf{El}\,\boldsymbol{a})\,\boldsymbol{B}$, note that the difference in presentation is exactly component-wise currying and uncurrying. For instance, in $\boldsymbol{t} : \mathbf{Tm}\,\Gamma\,(\boldsymbol{\Pi}\,\boldsymbol{a}\,\boldsymbol{B})$, the underlying action on contexts has the following type:

$$
(\underline{\Gamma} : \mathsf{Con}_{\boldsymbol{\Gamma}})(\gamma : \boldsymbol{a}\,\underline{\Gamma}) \to \mathsf{Con}_{\boldsymbol{B}}\,(\underline{\Gamma},\,\gamma)
$$

While in $\boldsymbol{t} : \mathbf{Tm}\,(\Gamma \rhd \mathbf{El}\,\boldsymbol{a})\,\boldsymbol{B}$, we have

$$
((\underline{\Gamma},\,\gamma) : (\underline{\Gamma} : \mathsf{Con}_{\boldsymbol{\Gamma}}) \times \boldsymbol{a}\,\underline{\Gamma}) \to \mathsf{Con}_{\boldsymbol{B}}\,(\underline{\Gamma},\,\gamma)
$$

So $\mathbf{app}$ and $\mathbf{lam}$ are defined as component-wise uncurrying and currying respectively. Naturality of $\boldsymbol{\Pi}$ and $\mathbf{app}$ again follows from the fact that flcwf morphisms strictly preserve all structure, and substitution is component-wise function composition.

External Function Type

For $Ix : \mathsf{Ty}_0$ and $\boldsymbol{B} : \mathsf{Tm}_0\, Ix \to \mathbf{Ty}\,\boldsymbol{\Gamma}$, we define $\boldsymbol{\Pi}^{\mathsf{ext}}\, Ix\, \boldsymbol{B} : \mathbf{Ty}\,\boldsymbol{\Gamma}$ as the Ix-indexed direct product of a family of displayed flcwfs.

$$
\begin{aligned}
\mathsf{Con}_{\boldsymbol{\Pi}^{\mathsf{ext}}\, Ix\, \boldsymbol{B}}\,\underline{\Gamma} &\quad :\equiv (i : \mathsf{Tm}_0\, Ix) \to \mathsf{Con}_{\boldsymbol{B}\, i}\,\underline{\Gamma} \\
\mathsf{Sub}_{\boldsymbol{\Pi}^{\mathsf{ext}}\, Ix\, \boldsymbol{B}}\,\Gamma\,\underline{\Delta}\,\underline{\sigma} &:\equiv (i : \mathsf{Tm}_0\, Ix) \to \mathsf{Sub}_{\boldsymbol{B}\, i}\,(\Gamma\, i)\,(\underline{\Delta}\, i)\,\underline{\sigma} \\
\mathsf{Ty}_{\boldsymbol{\Pi}^{\mathsf{ext}}\, Ix\, \boldsymbol{B}}\,\Gamma\,\underline{A} &\quad :\equiv (i : \mathsf{Tm}_0\, Ix) \to \mathsf{Ty}_{\boldsymbol{B}\, i}\,(\Gamma\, i)\,\underline{A} \\
\mathsf{Tm}_{\boldsymbol{\Pi}^{\mathsf{ext}}\, Ix\, \boldsymbol{B}}\,\Gamma\, A\,\underline{t} &\quad :\equiv (i : \mathsf{Tm}_0\, Ix) \to \mathsf{Tm}_{\boldsymbol{B}\, i}\,(\Gamma\, i)\,(A\, i)\,\underline{t}
\end{aligned}
$$

All structure is defined in the evident pointwise way. $\mathbf{app}^{\mathsf{ext}}$ and $\mathbf{lam}^{\mathsf{ext}}$ are defined by component-wise flipping of function arguments.

4.2.6 Discussion of Semantics

Flcwfs For Free

We give a quick summary for using the semantics of FQII signatures. As input we pick a) a signature Γ b) a cwf $\mathbb{C}$ with Σ, $\top$ and extensional Id. Then, we interpret the signature in $\mathbf{M}$, thereby getting an flcwf in 2LTT. Then, we interpret that in presheaves over $\mathbb{C}$, and we get the flcwf whose objects are internal Γ-algebras in $\mathbb{C}$.

One use case is in building models of certain type theories. Usually, this starts with constructing the base cwf. But if the objects can be specified using an FQII signature, we get an flcwf for free. In some cases, we get exactly what's needed. For example, the flcwf of presheaves can be used as it is in the presheaf models of type theories.

In other cases, the flcwf "for free" has to be extended with extra conditions. This often happens if the objects in the model have an internal notion of "equivalence" which has to be respected by types.

- In the setoid model, objects are setoids and types are displayed setoids with additional fibrancy structure [**?**].

- The groupoid model [HS96] is analogous; again types are displayed groupoids with fibrancy structure.

- Likewise, in the cubical set model [BCH14], types are displayed presheaves together with fibrancy structure (Kan composition).

In all these cases, the semantic objects have FQII signatures. We can take their flwcfs internally to **Set** and add fibrancy conditions. The cubical set model is presented exactly in this way in [BCH14], using displayed algebras. The groupoid model in [HS96] instead presents types as $\Gamma \to$ **Gpd** functors, i.e. uses an indexed style instead of the displayed style.

In the indexed-style groupoid model, we get strictly functorial type substitution, just like in the displayed style. However, the displayed style appears to be a more general way to get strict substitution, as it works for every FQII theory. Again, although finitely complete categories can be always strictified to cwfs, if we ever need to perform calculations with the internal definitions of a model, the displayed style is much more compact.

Variations of the Semantics

In Section 4.1, we required that the inner theory has Σ, $\top$ and extensional Id. Hence, when we interpret a signature, we again need to assume these type formers in $\mathbb{C}$. We used the assumed type formers in the definition of $\mathbf{U}$.

However, we can drop Id from the requirements on the inner theory, and likewise drop the identity type from flcwfs, and the model still works. In this case we have a somewhat more general semantics. In particular, like in Section 3.5.2, we can interpret signatures in finite product categories, because $\top$ and Σ can be derived from finite products in the constructed "simply typed" cwf. On the other hand, we get less out of the semantics. For instance, we cannot show equivalence of initiality and induction without Id.

We could also add more type formers to the semantics. We may add **small limits** via the external function type Π^{ext} that we already have in signatures. Extending flcwfs with Π^{ext} requires Π-types in the inner theory of 2LTT, hence in $\mathbb{C}$ as well. Intuitively, indexed products of algebras require functions in the underlying sorts. More concretely, in the definition of $\mathbf{U}$, we have to interpret

$$\Pi^{\mathsf{ext}}_{\mathbf{U}} : (Ix : \mathsf{Ty}_0) \to (\mathsf{Tm}_0\, Ix \to \mathsf{Ty}_{\mathbf{U}}\, \Gamma) \to \mathsf{Ty}_{\mathbf{U}}\, \Gamma$$

hence

$$\Pi^{\mathsf{ext}}_{\mathbf{U}} : (Ix : \mathsf{Ty}_0) \to (\mathsf{Tm}_0\, Ix \to \mathsf{Tm}_0\, \Gamma \to \mathsf{Ty}_0) \to \mathsf{Tm}_0\, \Gamma \to \mathsf{Ty}_0$$

This works if we can return an inner Π in the definition:

$$\Pi^{\mathsf{ext}}_{\mathbf{U}}\, Ix\, B\, \gamma :\equiv (i : Ix) \to B\, i\, \gamma$$

In this case, the "small limit" cwf semantics can be completed. We omit checking the details here. From Π^{ext}, we also recover indexed products, by using $\Pi^{\mathsf{ext}} Ix \, (\lambda i. \, \mathsf{K} \, \Gamma_i)$, where Γ_i is an indexed family of objects.

With the small limit semantics, if we want to have a simply typed interpretation, we can start with a cartesian closed $\mathbb{C}$.

Substitutions

Interpreting signatures is not the only potentially useful thing that we get out of the semantics. Each $\sigma : \mathsf{Sub} \, \Gamma \, \Delta$ can be viewed as a free interpretation of the Δ theory in Γ, and we get a strict flcwf morphism from the semantics.

One use case of Sub is to specify **ornaments** [**?**], i.e. ways to decorate structures with additional information, or dually, to erase parts of some structure. Ornaments differ from the usual forgetful maps, because they forget structure in *negative* position, i.e. in assumptions of construction rules.

Example 16. We assume $A : \mathsf{Ty}_0$. We define the substitution which forgets elements of A-lists.

$$\sigma : \mathsf{Sub} \, (\bullet \triangleright (Nat : \mathsf{U}) \triangleright (zero : \mathsf{El} \, Nat) \triangleright (suc : Nat))$$
$$(\bullet \triangleright (List : \mathsf{U}) \triangleright (nil : \mathsf{El} \, List) \triangleright (cons : A \to^{ext} List \to List))$$

The map goes from numbers to lists because of the "contravariant" forgetfulness. We define σ by listing its component definitions.

$$
\begin{aligned}
List \; &:\equiv Nat \\
nil \quad &:\equiv zero \\
cons \; &:\equiv \lambda^{ext} \, {}_{-}. \, \lambda \, n. \, suc \, n
\end{aligned}
$$

Example 17. We assume $\mathsf{Nat}_0 : \mathsf{Ty}_0$ with zero_0 and suc_0, and define $\sigma : \mathsf{Sub} \, \mathsf{NatSig} \, \mathsf{FinSig}$, where FinSig is as follows:

$$
\begin{aligned}
Fin \; &: \mathsf{Nat}_0 \to^{ext} \mathsf{U} \\
zero &: (n : \mathsf{Nat}_0) \to^{ext} \mathsf{El} \, (Fin \, (\mathsf{suc}_0 \, n)) \\
suc \; &: (n : \mathsf{Nat}_0) \to^{ext} Fin \, n \to \mathsf{El} \, (Fin \, (\mathsf{suc}_0 \, n))
\end{aligned}
$$

σ is defined as

$$
\begin{aligned}
Fin\ &:\equiv Nat \\
zero\ &:\equiv \lambda^{ext}\,\_.\,zero \\
suc\ &:\equiv \lambda^{ext}\,\_.\,\lambda\,n.\,suc\,n
\end{aligned}
$$

For a specific programming use case, if we have any recursive function defined on an "erased" type, we can convert that to a recursive function which acts on an "ornamented" type. For example, if we have some Nat-algebra Γ, the recursor yields a morphism from the initial algebra to Γ. We can map Γ to a list-algebra or a Fin-algebra, and then we can also use recursors for lists or Fin. Equivalently, we can map the unique morphism to Γ directly to a morphism between ornamented algebras.

Note though that a number of features and concepts from prior work on ornaments are not yet reproduced. For example, we don't yet have an analogue of *algebraic ornaments*, which would allow us produce an ornamented signature as an *output* of a generic operation, instead of assuming it to begin with. Exploring ornaments with QII signatures could be part of future work.

In a broader context, ToS provides a synthetic language for specifying **model constructions**. For example, we can show that constant families and democracy are equivalent, by defining an invertible Sub between signatures of cwf + K and cwf^dem, and getting an isomorphism of categories of models.

In many cases, this does not work, since ToS is rather restricted as a language. In Chapter 5, we present a more expressive ToS which is sufficient to rephrase many syntactic models [BPT17] as signature morphisms, thus morphisms of categories of models.

4.3 Recovering AMDS Interpretations

We have defined the **M** model in a "bundled" fashion, but sometimes we will also need to refer to pieces of it. On Figure 4.1 we have a summary of the model. On the left, the rows are labeled with components of ToS, while on the top we have components of flcwf. The individual rows can be further unfolded, as each of them contains multiple components. Likewise the Σ, Id and K columns can be unfolded. We get the whole model by filling in every cell of the unfolded table

	cwf					fl		
	Con	Sub	Ty	Tm	...	Σ	Id	K
cwf U Id Π Π^{ext}	$\_^A$	$\_^M$	$\_^D$	$\_^S$				

Figure 4.1: The flcwf model of the theory of signatures

with a definition. Of course, many of these cells are equations between equations, hence trivial by UIP.

This setup is very regular and convenient, because we can extract a displayed ToS model from any column, which may depend on displayed model to the left. The whole model is the total model of all columns. For example, the Con column doesn't depend on anything, so it's a plain model. The Ty column is displayed over Con. The Tm column is displayed over the total model of Con and Ty, but it does not actually depend on Sub.

From each displayed model, we get an eliminator, i.e. a family of interpretation functions. We note $-^A$, $-^M$, $-^D$ and $-^S$ in the table, but in principle we could refer to the eliminators of other columns as well. The interpretation functions can be defined in two ways:

- By separately taking the eliminators of each column, and referring to previous eliminators in each displayed model; e.g. referring to the eliminator maps $-^A$ in the definition of the Ty column.

- By taking the recursor for the entire model, and projecting out components from the result. E.g. we get $-^A$ by projecting out the first components of the interpretations of ToS objects.

However, the two versions coincide because of the initiality of ToS syntax.

Definition 44. From now on, we use $-^A$, $-^M$, $-^D$ and $-^S$ to refer to the families of functions that we described above.

4.4 Term Algebras

In this section we proceed with the construction of term algebras for FQII signatures, and their recursors and eliminators. We make some generalizations, in comparison to what we did with simple signatures.

- We assume that Ty_0 supports a model of ToS. As before, we use SigTy_i and SigTm_i in the following. Thus, we have $\mathsf{Con}_0 : \mathsf{Ty}_0$, $\mathsf{Sub}_0 : \mathsf{Con}_0 \to \mathsf{Con}_0 \to \mathsf{Ty}_0$, $\mathsf{SigTy}_0 : \mathsf{Con}_0 \to \mathsf{Ty}_0$ and $\mathsf{SigTm}_0 : (\Gamma : \mathsf{Con}_0) \to \mathsf{SigTy}_0\,\Gamma \to \mathsf{Ty}_0$. At the same time, we still have Con_1, Sub_1 and so on in the outer theory. Note that at this point, we *do not assume* any recursion or induction principle for the inner ToS model.

- We define lowering maps $\downarrow$ similarly as in Definition 38. Again, these maps preserve all structure.

Definition 45 (Term algebras). We fix a $\Omega : \mathsf{Con}_1$. We define $-^T$ by induction on the outer ToS. Formally, we need a displayed ToS model, but we instead present the resulting eliminator, which is perhaps easier to read. The underlying functions have the following types.

$$-^T : (\Gamma : \mathsf{Con}_1) \qquad (\nu : \mathsf{Sub}_0\,(\downarrow\Omega)\,(\downarrow\Gamma)) \to \Gamma^A$$
$$-^T : (A : \mathsf{SigTy}_1\,\Gamma) \quad (\nu : \mathsf{Sub}_0\,(\downarrow\Omega)\,(\downarrow\Gamma)) \to \mathsf{SigTm}_0\,(\downarrow\Omega)\,((\downarrow A)[\nu]) \to A^A\,(\Gamma^T\,\nu)$$
$$-^T : (\sigma : \mathsf{Sub}_1\,\Gamma\,\Delta) \quad (\nu : \mathsf{Sub}_0\,(\downarrow\Omega)\,(\downarrow\Gamma)) \to \Delta^T\,(\downarrow\sigma \circ \nu) \equiv \sigma^A\,(\Gamma^T\,\nu)$$
$$-^T : (t : \mathsf{SigTm}_1\,\Gamma\,A)(\nu : \mathsf{Sub}_0\,(\downarrow\Omega)\,(\downarrow\Gamma)) \to A^T\,\nu\,((\downarrow t)[\nu]) \equiv t^A\,(\Gamma^T\,\nu)$$

We review the idea of term algebras. In any model of ToS, we might think of a $\mathsf{Sub} \bullet \Gamma$ as a Γ-algebra internal to the model. In the $-^T$ interpretation we can assume $\Omega \equiv \bullet$; this means that from any internal Γ-algebra, in any model of ToS, we can extract an "external" Γ-algebra. This is possible because every sort $a : \mathsf{Tm}\,\Gamma\,\mathsf{U}$ in ToS induces an external set of terms as $\mathsf{Tm}\,\Gamma\,(\mathsf{El}\,a)$.

We can view the generalization from $\bullet$ to arbitrary Ω as switching from working in a given model M of ToS, to working in the *slice model* M/Ω, where contexts are Ω extended with zero or more entries. And in M/Ω, we have an Ω-algebra quite trivially, by taking the identity morphism $\mathsf{id} : \mathsf{Sub}\,\Omega\,\Omega^1$. Hence, term algebras

---

[1]$\mathsf{Sub}_{\mathsf{M}/\Omega} \bullet \Omega$ is isomorphic to, but not strictly the same as $\mathsf{Sub}_{\mathsf{M}}\,\Omega\,\Omega$.

arise by first taking the trivial internal algebra id in M/Ω, then converting it to an external algebra as $\Omega^T\,\mathsf{id} : \Omega^A$.

Remark. We could have presented $-^T$ and slice models in an explicit separate way. We instead chose to merge them into the current $-^T$, since we don't use slice models elsewhere, and we can skip their definition this way. Slices models would require the specification of *telescopes*, used to extend the base context, but this entails a fair amount of bureaucratic detail.

4.4.1   Induction for Term Algebras

4.4.2   Church Encoding

4.4.3   Awodey-Frey-Speight Encoding

4.5   Left Adjoints of Signature Morphisms

CHAPTER 5

Infinitary Quotient Inductive-Inductive Types

5.1 Theory of Signatures

5.2 Semantics

5.3 Term Algebras

CHAPTER 6

Levitation, Bootstrapping and Universe Levels

6.1   Levitation for Closed QIITs

6.2   Levitation for Infinitary QIITs

CHAPTER 7

Higher Inductive-Inductive Types

7.1 Theory of Signatures

7.2 Semantics

CHAPTER 8

Reductions

8.1   Finitary Inductive Types

8.2   Finitary Inductive-Inductive Types

8.3   Closed Quotient Inductive-Inductive Types

CHAPTER 9

Conclusion

Bibliography

[AAC⁺20] Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. A type and scope safe universe of syntaxes with binding: Their semantics and proofs. *CoRR*, abs/2001.11001, 2020.

[ACD⁺18] Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall Forsberg. Quotient inductive-inductive types. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10803 of *Lecture Notes in Computer Science*, pages 293–310. Springer, 2018.

[ACKS19] Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. Two-level type theory and applications. *ArXiv e-prints*, may 2019.

[AK79] Jirí Adámek and Václav Koubek. Least fixed point of a functor. *J. Comput. Syst. Sci.*, 19(2):163–178, 1979.

[AL19] Benedikt Ahrens and Peter LeFanu Lumsdaine. Displayed Categories. *Logical Methods in Computer Science*, Volume 15, Issue 1, March 2019.

[AMM19] Benedikt Ahrens, Ralph Matthes, and Anders Mörtberg. From signatures to monads in unimath. *J. Autom. Reason.*, 63(2):285–318, 2019.

[AVW17] Andreas Abel, Andrea Vezzosi, and Théo Winterhalter. Normalization by evaluation for sized dependent types. *Proc. ACM Program. Lang.*, 1(ICFP):33:1–33:30, 2017.

[Awo18]   Steve Awodey.   Natural models of homotopy type theory.   *Math. Struct. Comput. Sci.*, 28(2):241–286, 2018.

[BCH14]   Marc Bezem, Thierry Coquand, and Simon Huber. A model of type theory in cubical sets. In *19th International Conference on Types for Proofs and Programs (TYPES 2013)*, volume 26, pages 107–128, 2014.

[BH11]   Patrick Bahr and Tom Hvitved. Compositional data types. In Jaakko Järvi and Shin-Cheng Mu, editors, *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming, WGP@ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 83–94. ACM, 2011.

[BKS21]   Rafaël Bocquet, Ambrus Kaposi, and Christian Sattler. Induction principles for type theories, internally to presheaf categories. *arXiv preprint arXiv:2102.11649*, 2021.

[BPT17]   Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. The next 700 syntactical models of type theory. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, pages 182–194, New York, NY, USA, 2017. ACM.

[Cap17]   Paolo Capriotti. Models of type theory with strict equality. *arXiv preprint arXiv:1702.04912*, 2017.

[Car78]   John Cartmell. *Generalised algebraic theories and contextual categories.* PhD thesis, Oxford University, 1978.

[CD14]   Pierre Clairambault and Peter Dybjer. The biequivalence of locally cartesian closed categories and martin-löf type theories. *Mathematical Structures in Computer Science*, 24(6), 2014.

[dVL14]   Edsko de Vries and Andres Löh. True sums of products. In José Pedro Magalhães and Tiark Rompf, editors, *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming, WGP 2014, Gothenburg, Sweden, August 31, 2014*, pages 83–94. ACM, 2014.

[Dyb95]   Peter Dybjer. Internal type theory. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers*, vol-

ume 1158 of *Lecture Notes in Computer Science*, pages 120–134. Springer, 1995.

[GKNB20] Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. Multimodal dependent type theory. In Holger Hermanns, Lijun Zhang, Naoki Kobayashi, and Dale Miller, editors, *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*, pages 492–506. ACM, 2020.

[HRR14] Claudio Hermida, Uday S. Reddy, and Edmund P. Robinson. Logical relations and parametricity - A reynolds programme for category theory and programming languages. *Electron. Notes Theor. Comput. Sci.*, 303:149–180, 2014.

[HS96] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In *In Venice Festschrift*, pages 83–111. Oxford University Press, 1996.

[Hug21] Jasper Hugunin. Why not w?, 2021.

[Joh02] Peter T Johnstone. *Sketches of an elephant: A topos theory compendium*, volume 2. Oxford University Press, 2002.

[Kov21] András Kovács. Generalized universe hierarchies and first-class universe levels, 2021.

[KvR20] Ambrus Kaposi and Jakob von Raumer. A syntax for mutual inductive families. In Zena M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference)*, volume 167 of *LIPIcs*, pages 23:1–23:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[LM11] Andres Löh and José Pedro Magalhães. Generic programming with indexed functors. In Jaakko Järvi and Shin-Cheng Mu, editors, *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming, WGP@ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 1–12. ACM, 2011.

[MFP91]   Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer, 1991.

[nc21]   nLab contributors. Internal logic, 2021.

[NF13]   Fredrik Nordvall Forsberg. *Inductive-inductive definitions*. PhD thesis, Swansea University, 2013.

[PWK19]   Matthew Pickering, Nicolas Wu, and Csongor Kiss. Multi-stage programs in context. In Richard A. Eisenberg, editor, *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18-23, 2019*, pages 71–84. ACM, 2019.

[Sch17]   Gabriel Scherer. Deciding equivalence with sums and the empty type. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 374–386. ACM, 2017.

[Swi08]   Wouter Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, 2008.

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

[YHLJ09] Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring. Generic programming with fixed points for mutually recursive datatypes. In Graham Hutton and Andrew P. Tolmach, editors, *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, pages 233–244. ACM, 2009.