Type-Theoretic Signatures for 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 in the current chapter, 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.

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

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

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.

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

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

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

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.

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

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

$$\begin{aligned} \mathsf{NatSig} &\equiv \bullet \triangleright (\mathit{zero} : \iota) \triangleright (\mathit{suc} : \iota \to \iota) \\ \mathsf{TreeSig} &\equiv \bullet \triangleright (\mathit{leaf} : \iota) \triangleright (\mathit{node} : \iota \to \iota \to \iota) \end{aligned}$$

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, $\mathit{node} : (\mathbb{N} \to \iota) \to \iota$ 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 the terms "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.

$$
\begin{aligned}
&-^A : \mathsf{Ty} \to \mathsf{Set} \to \mathsf{Set}\\
&\iota^A \qquad\quad X \equiv X\\
&(\iota \to A)^A\, X \equiv X \to A^A\, X
\end{aligned}
$$

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

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

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

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 $(\mathit{zero} : X) \times (\mathit{suc} : X \to X)$ 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 $(X : \mathsf{Set}) \times (\mathit{zero} : X) \times (\mathit{suc} : X \to X)$.

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:

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

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:

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

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

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

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

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. We again pack up Γ^M with the interpretation of ι, to get notions of *algebra morphisms*:

$$\begin{aligned}&\mathsf{Mor} : \{\Gamma : \mathsf{Con}\} \to \mathsf{Alg}\,\Gamma \to \mathsf{Alg}\,\Gamma \to \mathsf{Set}\\&\mathsf{Mor}\,\{\Gamma\}\,(X_0,\,\gamma_0)\,(X_1,\,\gamma_1) \equiv (X^M : X_0 \to X_1) \times \Gamma^M\,X^M\,\gamma_0\,\gamma_1\end{aligned}$$

Example 3. We have the following computation:

$$\begin{aligned}&\mathsf{Mor}\,\{\mathsf{NatSig}\}\,(X_0,\,\mathit{zero}_0,\,\mathit{suc}_0)\,(X_0,\,\mathit{zero}_1,\,\mathit{suc}_1) \equiv\\&\qquad (X^M : X_0 \to X_1)\\&\qquad \times (X^M\,\mathit{zero}_0 = \mathit{zero}_1)\\&\qquad \times ((x : X_0) \to X^M\,(\mathit{suc}_0\,x) = \mathit{suc}_1\,(X^M\,x))\end{aligned}$$

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

$$\begin{aligned}&\mathsf{Initial} : \{\Gamma : \mathsf{Con}\} \to \mathsf{Alg}\,\Gamma \to \mathsf{Set}\\&\mathsf{Initial}\,\{\Gamma\}\,\gamma \equiv (\gamma' : \mathsf{Alg}\,\Gamma) \to \mathsf{isContr}\,(\mathsf{Mor}\,\Gamma\,\gamma\,\gamma')\end{aligned}$$

Here isContr refers to unique existence [Uni13, Section 3.11]. If we drop isContr from the definition, we get the notion of weak initiality, which corresponds to the recursion principle for Γ. Although we call this predicate 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:

$$\begin{aligned}&-^R : (A : \mathsf{Ty})\{X_0\,X_1 : \mathsf{Set}\}(X^R : X_0 \to X_1 \to \mathsf{Set}) \to A^A\,X_0 \to A^A\,X_1 \to \mathsf{Set}\\&\iota^R \qquad\quad X^R\,\alpha_0\;\alpha_1 \equiv X^R\,\alpha_0\,\alpha_1\\&(\iota \to A)^R\,X^R\,\alpha_0\;\alpha_1 \equiv (x_0 : X_0)(x_1 : X_1) \to X^R\,x_0\,x_1 \to A^R\,X^R\,(\alpha_0\,x_0)\,(\alpha_1\,x_1)\end{aligned}$$

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.

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

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

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

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

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)**.**

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

Example 4. We have the following computation.

$$
\begin{aligned}
&\mathsf{DispAlg}\, \{\mathsf{NatSig}\}\, (X,\, \mathit{zero},\, \mathit{suc}) \equiv\\
&\qquad (X^D \quad : X \to \mathsf{Set})\\
&\qquad \times (\mathit{zero}^D : X^D\, \mathit{zero})\\
&\qquad \times (\mathit{suc}^D \ : (n : X) \to X^D\, n \to X^D\, (\mathit{suc}\, n))
\end{aligned}
$$

2.2.4 Sections

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

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

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

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

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

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.

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

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 DispAlg is a bundle of the induction motive and the methods, and the output 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.

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

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

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

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

Now we can define the term algebra for Ω itself:

$$
\begin{aligned}
&\mathsf{TmAlg}_\Omega : \mathsf{Alg}\,\Omega\\
&\mathsf{TmAlg}_\Omega \equiv \Omega^T\,\Omega\,\mathsf{id}
\end{aligned}
$$

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 substitutions is analogous.

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 4.3 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.

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

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

We define the recursor for Ω as

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

In short, the way we get recursion is by evaluating terms in arbitrary (X, ω) 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:

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

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:

$$
\begin{aligned}
&-^E : (A : \mathsf{Ty})(t : \mathsf{Tm}\,\Omega\,A) \to A^S\,\mathsf{E}\,(t^A\,(\Omega^T\,\Omega\,\mathsf{id}))\,(t^D\,\omega^D)\\
&\iota^E \qquad\quad t : (t^A\,(\Omega^T\,\Omega\,\mathsf{id}))^D\,\omega^D = t^D\,\omega^D\\
&(\iota \to A)^E\,t \equiv \lambda\,u.\,A^E\,(\mathsf{app}\,t\,u)
\end{aligned}
$$

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

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.

$$
\begin{aligned}
&\mathsf{Ind}_\Omega : (alg : \mathsf{DispAlg}\,\mathsf{TmAlg}_\Omega) \to \mathsf{Section}\,\mathsf{TmAlg}_\Omega\,alg\\
&\mathsf{Ind}_\Omega\,(X^D,\,\omega^D) \equiv (E,\,\Omega^E\,\Omega\,\mathsf{id})
\end{aligned}
$$

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 **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 a 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 $\mathsf{K}\top$ codes the functor which is constantly $\top$. $-\otimes-$ and $-\oplus-$ are pointwise products and coproducts respectively. So we have the evident interpretation functions:

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

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

$$\begin{aligned}&\mathsf{Fix}_\mathsf{F} \;: \mathsf{Set}\\ &\mathsf{con}_\mathsf{F} : [\![F]\!]\, \mathsf{Fix}_\mathsf{F} \to \mathsf{Fix}_\mathsf{F}\end{aligned}$$

In Haskell, this definition is valid for arbitrary *semantic* F functor, because there is no termination checking, and thus no positivity checking. In Agda, the above definition is valid because the positivity checker looks inside the definition of $[\![-]\!]$ and lets it through. Next we establish weak initiality, by defining the recursor:

$$\begin{aligned}&\mathsf{rec} : ([\![F]\!]\, X \to X) \to \mathsf{Fix}_\mathsf{F} \to X\\ &\mathsf{rec}\, f\, (\mathsf{con}\, x) \equiv f\, (\mathsf{map}\, (\mathsf{rec}\, f)\, x)\end{aligned}$$

This is again fine in Haskell, but it unfortunately does not pass Agda's termination checker. There are several possible solutions, assuming that we stick to functors as signatures:

1. Using *sized types* [AVW17]. For example, this was used in [AAC$^+$20]. The drawback is dependence on an extra language feature which is only supported in Agda, out of the major dependently typed languages.

2. Turning termination checking off.

3. Not using the direct inductive definition for fixpoints, and thereby not being exposed to the whims of syntactical strict positivity checking in Agda. Instead, 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 of this approach 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:

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

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 term algebras described in this chapter compare quite favorably. As we have seen, induction for term algebras can be defined in a small amount of easy code, without using sized types or quotients.

For practical usefulness, it makes sense to make a slight modification of terms. We switch to a *spine neutral* definition. We mutually inductively define Spine and

Tm:

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

With this representation, a term is always a variable applied to a list of arguments. This can be useful, because the pattern matching implementation of the metalanguage knows about which constructors are possible, when matching on values of $\mathsf{Tm}\,\Gamma\,A$. Using this, we give here an ad-hoc definition of natural numbers in pseudo-Agda.

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

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

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.

formalize Rec and Ind with spines in Agda

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)? Informally, it is easy to see that any concrete definition of an eliminator like $\mathsf{NatElim}$ enjoys definitional β. In this chapter we do not have a nice way of reasoning about definitional equalities. In the next chapter we develop such reasoning and show that an Ind_Ω which is defined internally to an intensional theory (as in this chapter) only has propositional β-rules, but if it is defined in a metaprogramming layer, it has definitional β, even if we don't use spine neutral terms.

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].

The closest 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.

Hence, it remains future work to see how term algebras might be used in more practically-oriented generic programming.

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. Then, we introduce two-level type theory more formally and describe its intended models. Finally, we generalize the semantics and the term algebra construction from Chapter 2 in type-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, 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 and elimination with strict β-rules, and for this we need to reason about strict equality.

Second, 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 intensional type theory.

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

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 $(-,-) : \mathbb{C}(A, B) \to \mathbb{C}(A, C) \to \mathbb{C}(A, B \otimes C)$ and π_1 and π_2 for projections.

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 an unspecified metatheory, and the $\times$ in the definitions refer to the metatheoretic Σ. The algebras can be viewed as diagrams which preserve finite products, and algebra morphisms become 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 2LTT

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.

3.2.1 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 look at models of type theories 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.

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:

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

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

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

Definition 13. The following derivable operations are commonly required.

- *Single substitution* can be derived from parallel substitution as follows. Assume $t : \mathsf{Tm}\,(\Gamma \triangleright A)\,B$, and $u : \mathsf{Tm}\,\Gamma\,A$. 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 $t[\mathsf{id}, u] : \mathsf{Tm}\,\Gamma\,(\mathsf{A}[\mathsf{id}, u])$. Note that term substitution causes the type to be substituted as well. $(\mathsf{id}, u) : \mathsf{Sub}\,\Gamma\,(\Gamma \triangleright A)$ is well-typed because $u : \mathsf{Tm}\,\Gamma\,A$ hence also $u : \mathsf{Tm}\,\Gamma\,(A[\mathsf{id}])$.

- We can *lift substitutions* over binders as follows. Assuming $\sigma : \mathsf{Sub}\,\Gamma\,\Delta$ and $A : \mathsf{Ty}\,\Delta$, we construct a lifting of σ which maps an additional A-variable to itself: $(\sigma \circ \mathsf{p}, \mathsf{q}) : \mathsf{Sub}\,(\Gamma \triangleright A[\sigma])\,(\Delta \triangleright A)$. Let us see why this is well-typed. We have $\mathsf{p} : \mathsf{Sub}\,(\Gamma \triangleright A[\sigma])\,\Gamma$ and $\sigma : \mathsf{Sub}\,\Gamma\,\Delta$, so $\sigma \circ \mathsf{p} : \mathsf{Sub}\,(\Gamma \triangleright A[\sigma])\,\Delta$. Also, $\mathsf{q} : \mathsf{Tm}\,(\Gamma \triangleright A[\sigma])\,(A[\sigma][\mathsf{p}])$, hence $\mathsf{q} : \mathsf{Tm}\,(\Gamma \triangleright A[\sigma])\,(A[\sigma \circ \mathsf{p}])$, 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 $t : \mathsf{Tm}\,\Gamma\,A$, 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, $t : \mathsf{Tm}\,(\Gamma \triangleright x : B)\,A$ may be used.[2]

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

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

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 can 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.

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

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* is sometimes used for this style. It lets us also omit contexts, so we would only need to write

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

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. One more reason, and perhaps the most important one, for skipping higher-order abstract syntax, is that this author has not yet acquired the proficiency to comfortably use the new techniques.

Definition 15 (Σ-types)**.** A family structure supports Σ-types if we have

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

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 16 (Unit type)**.** 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.

The identity type can be also specified as a negative type. In this case we get an extensional identity, while if we define it as a positive type, we get intensional identity instead.

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.

Definition 17 (Extensional identity)**.** 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 $(\mathsf{reflect}, \mathsf{refl}) : \mathsf{Tm}\,\Gamma\,(\mathsf{Id}\,t\,u) \simeq (t = u)$.

$\mathsf{refl} : t = u \to \mathsf{Tm}\,\Gamma\,(\mathsf{Id}\,t\,u)$ 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 set in the metatheory (by assumption, or by not having higher-dimensional types 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.

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

$\mathsf{BoolRec}$ 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 $m : \mathsf{Tm}\,(\Gamma \triangleright b : \mathsf{Bool})\,B$, such that $m[b \mapsto \mathsf{true}] = t$ and $m[b \mapsto \mathsf{false}] = f$, it follows that $\mathsf{BoolRec}\,B\,t\,f\,b : \mathsf{Tm}\,(\Gamma \triangleright b : \mathsf{Bool})\,B$ 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 of basic intensional type theory, since now we would have equality proofs besides 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

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

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

TODO

CHAPTER 4

Finitary Quotient Inductive-Inductive Types

4.1 Theory of Signatures

4.1.1 Models

4.1.2 Examples

4.2 Semantics

4.2.1 Finite Limit Cwfs

4.2.2 Equivalence of Initiality and Induction

4.2.3 Model of the Theory of Signatures

4.3 Term Algebras

4.3.1 Generic Term Algebras

4.3.2 Induction for Term Algebras

4.3.3 Church Encoding

4.3.4 Awodey-Frey-Speight Encoding

4.4 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.

[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.

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

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

[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.

[Hug21] Jasper Hugunin. Why not w?, 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 pro-*

gramming, WGP@ICFP 2011, Tokyo, Japan, September 19-21, 2011, pages 1–12. ACM, 2011.

[nc21] nLab contributors. Internal logic, 2021.

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

[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.