Notes on Type Theory

[DRAFT: March 23, 2025]

Steve Awodey

with contributions from Andrej Bauer

Contents

Chapter 3

Dependent Type Theory

The Curry-Howard correspondence from Chapter **??** can be extended to natural deduction proofs in *first-order* logic, providing an extension of the "propositions as types/proofs as terms" idea from propositional logic to first-order logic (see [Sco70, How80] . In addition to simple types $A, B, ...$ representing propositions, one then has *dependent* types $x : A \vdash B(x)$ representing "propositional functions" or predicates. In addition to the simple type formers $A \times B$ and $A \to B$, one has dependent type formers $\Sigma_{x:A}B(x)$ and $\Pi_{x:A}B(x)$, representing the quantified propositions $\exists_{x:A}B(x)$ and $\forall_{x:A}B(x)$. As before, these types may have different terms $s, t : \Pi_{x:A}B(x)$, resulting from different proofs of the corresponding propositions, so that the calculus of terms again records more information than mere *provability.* Also as before, the resulting abstract structure turns out to be one that is shared by other categories *not* arising from logic—and now the coincidence is even more remarkable, because the structure at issue is a much more elaborate one. Where proofs in the propositional calculus gave rise to a cartesian closed category, the category of proof terms of first-order logic will be seen to be *locally cartesian closed*, a mathematical structure also shared by sheaves on a space, Grothendieck toposes, categories of fibrations, and other important examples.

Before stating a formal dependent type theory, we begin by infomally "categorifying" first-order logic with an abstraction (due to Lawvere [Law70]) called a *hyperdoctrine.* A hyperdoctrine is a contravariant functor $P : \mathcal{C}^{\mathsf{op}} \to \mathsf{Cat}$ (see Section 3.1), and there are in particular both poset-valued and "proper" category-valued ones. The former correspond to propositional and predicate logic, while the latter correspond more closely to dependent type theory, where the individual value categories $P(C)$ may be proper cartesian closed categories (rather than just Heyting algebras or CCC posets). Moreover, the reindexing functors along all projections $p_A : X \times A \to A$ in the index category $\mathcal{C}$ of contexts are also required to admit both left and right adjoints $\Sigma_A \dashv p_A^* \dashv \Pi_A$, according to Lawvere's adjoint analysis of quantification. An important *difference* between hyperdoctrines and dependent type theories, however, is that the indexing category of contexts in dependent type theory has not just finite products, but also some additional structure resulting from an operation of *context extension*, which takes as input a type in context $\Gamma \vdash A$ and returns a new context $(\Gamma, x : A)$, together with a substitution arrow $(\Gamma, x : A) \to \Gamma$. This is taking the "propositions-as-types" idea even more seriously, by allowing every proposition $\Gamma \vdash \varphi$ in first-order logic to form a new type $\{\Gamma \vdash \varphi\}$, thus turning the objects $A \in P(C)$ in the value-categories of hyperdoctrine $(\mathcal{C}, P)$ into arrows $\{A\} \to C$ in $\mathcal{C}$.[1]

3.1 Hyperdoctrines

Given an algebraic signature, let $\mathcal{C}$ be the category of contexts, with (non-dependent) tuples of typed variables $\Gamma = (x_1 : C_1, ..., x_n : C_n)$ as objects, and as arrows $\gamma : \Delta \to \Gamma$ the n-tuples of terms $c_1 : C_1, \ldots, c_n : C_n$, all in context $\Delta = (y_1 : D_1, ..., y_m : D_m)$,

$$\Delta \vdash c_i : C_i\,, \quad 1 \leq i \leq n\,.$$

Composition is given by substitution of terms for variables,

$$\gamma \circ \delta = (c_1[d_1/y_1, \ldots, d_m/y_m], \ldots, c_n[d_1/y_1, \ldots, d_m/y_m],)$$

for $\delta = (d_1, \ldots, d_m) : \mathsf{E} \to \Delta$ with $\mathsf{E} = (z_1 : E_1, ..., z_k : E_k)$, and the identity arrows are the variables themselves (terms are identified up to α-renaming of variables, as in Lawvere algebraic theories, see Chapter **??**). The category $\mathcal{C}$ then has all finite products, essentially given by tupling.

For each object Γ, let $P(\Gamma)$ be the *poset* of all first-order formulas $(\Gamma \mid \varphi)$, ordered by entailment $\Gamma \mid \varphi \vdash \psi$ and identified up to provable equivalence $\Gamma \mid \varphi \dashv\vdash \psi$. Substitution of a term $\sigma : \Delta \to \Gamma$ into a formula $(\Gamma \mid \varphi)$ then determines a morphism of posets $\sigma^* : P(\Gamma) \to P(\Delta)$, which also preserves all of the propositional operations,

$$\sigma^*(\varphi \wedge \psi) = \varphi[\sigma/x] \wedge \psi[\sigma/x] = \sigma^*(\varphi) \wedge \sigma^*(\psi), \qquad \text{etc.}$$

(Exercise!). Moreover, since substitutions into formulas and terms commute with each other, $\tau^*\sigma^*\varphi = \varphi[\sigma \circ \tau/x]$, this action is *strictly* functorial, and so we have a contravariant functor

$$P : \mathcal{C}^{\mathsf{op}} \longrightarrow \mathsf{Heyt}$$

from the category of contexts to the category of Heyting algebras.

Now consider the quantifiers $\exists$ and $\forall$. Given a projection of contexts $p_X : \Gamma \times X \to \Gamma$, in addition to the pullback functor

$$p_X^* : P(\Gamma) \longrightarrow P(\Gamma \times X)$$

induced by weakening, there are the operations of quantification

$$\exists_X, \forall_X : P(\Gamma \times X) \longrightarrow P(\Gamma)\,.$$

By the rules for the quantifiers, these are indeed left and right adjoints to weakening,

$$\exists_X \dashv p_X^* \dashv \forall_X\,.$$

The Beck-Chevalley rules assert that substitution commutes with quantification, in the sense that $(\forall_x\varphi)[s/y] = \forall_x(\varphi[s/y])$, and similarly for $(\exists_x\varphi)$.

[1] [Law70] does just this.

Definition 3.1.1. A *(posetal) hyperdoctrine* consists of a Cartesian category $\mathcal{C}$ together with a contravariant functor

$$P : \mathcal{C}^{\mathsf{op}} \longrightarrow \mathsf{Heyt}\,,$$

such that for each $f : D \to C$ the action maps $f^* = Pf : PC \to PD$ have both left and right adjoints

$$\exists_f \dashv f^* \dashv \forall_f$$

that satisfy the Beck-Chavalley conditions.

Exercise 3.1.2. Verify that the syntax of first-order logic can indeed be organized into a hyperdoctrine in the way just described.

Examples

1. We just described the syntactic example of first-order logic. Indeed, for each first-order theory $\mathbb{T}$ there is an associated hyperdoctrine $(\mathcal{C}_{\mathbb{T}}, P_{\mathbb{T}})$, with the types and terms of $\mathbb{T}$ as the category of contexts $\mathcal{C}_{\mathbb{T}}$, and the formulas (in context) of $\mathbb{T}$ as "predicates", i.e. the elements of the Heyting algebras $\varphi \in P_{\mathbb{T}}(\Gamma)$. A general hyperdoctrine can be regarded as an abstraction of this example.

2. A hyperdoctrine on the index category $\mathcal{C} = \mathsf{Set}$ is given by the powerset functor

   $$\mathcal{P} : \mathsf{Set}^{\mathsf{op}} \longrightarrow \mathsf{Heyt}\,,$$

   which is represented by the Heyting algebra $\mathbb{2}$, in the sense that for each set I one has

   $$\mathcal{P}(I) \cong \mathsf{Hom}(I, \mathbb{2})\,.$$

   Similarly, for any *complete* Heyting algebra H in place of $\mathbb{2}$, there is a hyperdoctrine H-Set, with

   $$P_{\mathsf{H}}(I) \cong \mathsf{Hom}(I, \mathsf{H})\,.$$

   The adjoints to precomposition along a map $f : J \to I$ are given by

   $$\begin{aligned} \exists_f(\varphi)(i) &= \bigvee_{j\in J} (f(j) = i) \wedge \varphi(j)\,, \\ \forall_f(\varphi)(i) &= \bigwedge_{j\in J} (f(j) = i) \Rightarrow \varphi(j)\,, \end{aligned}$$

   where the value of $x = y$ in H is defined to be $\bigvee\{\top \mid x = y\}$.

   We leave it as an exercise to verify that this is hyperdoctrine, in particular to show that the Beck-Chevalley conditions are satisfied.

   Exercise 3.1.3. Show this.

3. For a related example, let $\mathbb{C}$ be any small index category and $\mathcal{C} = \widehat{\mathbb{C}}$, the category of presheaves on $\mathbb{C}$. An internal Heyting algebra H in $\mathcal{C}$, i.e. a functor $\mathbb{C}^{\mathsf{op}} \to \mathsf{Heyt}$, is said to be *internally complete* if, for every $I \in \mathcal{C}$, the transpose $\mathsf{H} \to \mathsf{H}^I$ of the projection $\mathsf{H} \times I \to \mathsf{H}$ has both left and right adjoints. Such an internally complete Heyting algebra determines a (representable) hyperdoctrine $P_{\mathsf{H}} : \mathcal{C} \to \mathsf{Set}$ just as for the case of $\mathcal{C} = \mathsf{Set}$, by setting $P_{\mathsf{H}}(C) = \mathcal{C}(C, \mathsf{H})$.

4. For any Heyting category $\mathcal{H}$ let $\mathsf{Sub}(C)$ be the Heyting algebra of all subobjects $S \rightarrowtail C$ of the object C. The presheaf $\mathsf{Sub} : \mathcal{H}^{\mathsf{op}} \to \mathsf{Heyt}$, with action by pullback, is then a hyperdoctrine, essentially by the definition of a Heyting category.

Remark 3.1.4 (Lawvere's Law)**.** In any hyperdoctrine $(\mathcal{C}, P)$, for each object $C \in \mathcal{C}$, we can determine an equality relation $=_C$ in each $P(C \times C)$, namely by setting

$$(x =_C y) \;=\; \exists_{\Delta_C}(\top)\,,$$

where $\Delta_C : C \to C \times C$ is the diagonal, $\exists_{\Delta_C} \dashv \Delta_C^*$, and $\top \in P(C)$. Displaying variables for clarity, if $\rho(x,y) \in P(C \times C)$ then $\Delta_C^*\rho(x,y) = \rho(x,x) \in PC$ is the contraction of the different variables, and the adjunction $\exists_{\Delta_C} \dashv \Delta_C^*$ can be formulated as the following two-way rule,

$$\frac{x : C \mid \top \vdash \rho(x,x)}{x : C, y : C \mid (x =_C y) \vdash \rho(x,y)} \tag{3.1}$$

which expresses that $(x =_C y)$ is the least reflexive relation on C. See [Law70] and Exercise **??** above.

Exercise 3.1.5. Prove the standard first-order laws of equality from the above hyperdoctrine formulation of Lawvere's Law (3.1).

Proper hyperdoctrines

Now let us consider some hyperdoctrines of a different kind. For any set I, let Set^I be the category of *families of sets* $(A_i)_{i \in I}$, with families of functions $(g_i : A_i \to B_i)_{i \in I}$ as arrows, and for $f : J \to I$ let us reindex along f by the precomposition functor $f^* : \mathsf{Set}^I \to \mathsf{Set}^J$, with

$$f^*((A_i)_{i \in I})_j = A_{f(j)}\,.$$

Thus we have a contravariant functor

$$P : \mathsf{Set}^{\mathsf{op}} \to \mathsf{Cat}$$

with $P(I) = \mathsf{Set}^I$ and $f^*(A : I \to \mathsf{Set}) = A \circ f : J \to \mathsf{Set}$.

Lemma 3.1.6. *The precomposition functors $f^* : \mathsf{Set}^I \to \mathsf{Set}^J$ have both left and right adjoints $f_! \dashv f^* \dashv f_*$ which can be computed by the formulas:*

$$f_!(A)_i = \coprod_{j \in f^{-1}\{i\}} A_j \,, \tag{3.2}$$
$$f_*(A)_i = \prod_{j \in f^{-1}\{i\}} A_j \,,$$

for $A = (A_j)_{j \in J}$. Moreover, these functors satisfy the Beck-Chevally conditions.

A closely related example uses the familiar equivalence of categories $\mathsf{Set}^I \simeq \mathsf{Set}/_I$, where now the adjoints

$$f_! \dashv f^* \dashv f_* : \mathsf{Set}/_J \longrightarrow \mathsf{Set}/_I$$

to reindexing along $f : J \to I$ are (post-)composition, pullback, and "push-forward", respectively. In this case, the action of the pseudofunctor P is not strictly functorial, as it was for the case of $P(I) = \mathsf{Set}^I$. Note that the Beck-Chevalley conditions for such Cat-valued functors should now also be stated as (canonical) isomorphisms, rather than equalities as they were for poset-valued functors. In this way, when the individual categories $P(I)$ are proper, and not just posets, the entire hyperdoctrine structure may be weakened to include (coherent) isomorphisms, both in the functorial action of P, and in the B-C conditions. We will not spell out the required coherences here, but the interested reader may look up the corresponding notion of an *indexed-category*, which is a Cat-valued *pseudofunctor* (see [Joh03, B1.2]).

Example 3.1.7. Another example of a "proper" hyperdoctrine, with values in non-posetal (large!) categories, is the category of presheaves construction $\widehat{\mathbb{C}} = \mathsf{Set}^{\mathbb{C}^{\mathsf{op}}}$, where:

$$\mathcal{P} : \mathsf{Cat}^{\mathsf{op}} \longrightarrow \mathsf{CAT}\,,$$
$$\mathbb{C} \longmapsto \widehat{\mathbb{C}}\,.$$

Here the action of $\mathcal{P}$ may be assumed to be *strictly* functorial, because it's given by precomposition. Nonetheless the B-C conditions must be stated as natural isos, because the adjoints $F_! \dashv F^* \dashv F_* : \widehat{\mathbb{D}} \longrightarrow \widehat{\mathbb{C}}$ for $F : \mathbb{D} \to \mathbb{C}$ are given by left and right Kan extensions, which need not be strictly functorial.

We shall consider several more examples of proper hyperdoctrines below. The internal logic of such categories generalizes and "categorifies" first-order logic, and is better described as dependent type theory. Proper hyperdoctrines $P : \mathcal{C}^{\mathsf{op}} \to \mathsf{Cat}$ are roughly related to dependent type theory in the way that posetal ones $P : \mathcal{C}^{\mathsf{op}} \to \mathsf{Pos}$ are related to FOL. There are actually two distinct aspects of this generalization: (1) the individual categories of "predicates" $P(C)$ are proper categories rather than mere posets, (2) the variation over the index category $\mathcal{C}$ of contexts (and its adjoints) is weakened accordingly to pseudo-functoriality. Each of these aspects plays an important role in dependent type theory and its categorical semantics.

First-Order Logic	Dependent Type Theory
Propositional Logic	Simple Type Theory

3.2 Dependently-typed lambda-calculus.

We give a somewhat informal specification of the syntax of the *dependently-typed λ-calculus* (see [Hof95, AG] for a more detailed exposition).

Dependent type theories have four standard forms of judgement

$$A : \mathtt{type}\,, \quad A \equiv B : \mathtt{type}\,, \quad a : A\,, \quad a \equiv b : A\,.$$

We refer to the triple equality relation $\equiv$ in these judgements as *definitional (or judgemental) equality.* It should not be confused with the notions of *(extensional and intensional) propositional equality* to be introduced below. A judgement J of one of the four above kinds can also be made relative to a *context* Γ of variable declarations, a situation that we indicate by writing $\Gamma \vdash J$. When stating deduction rules for such judgements we make use of standard conventions to simplify the exposition, such as omitting the (part of the) context that is common to premisses and conclusions of the rule.

To formulate the rules, we revisit the rules of simple type theory from Section **??** and adjust them as follows.

Judgements: The basic kinds of judgements are:

$$\Gamma\ \mathtt{ctx}\ , \qquad \Gamma \vdash A\ \mathtt{type}\ , \qquad \Gamma \vdash a : A\,.$$

along with the judgemental equalities of each kind:

$$\Gamma \equiv \Delta\ \mathtt{ctx} \qquad A \equiv B\ \mathtt{type} \qquad a \equiv b : A\,,$$

each of which are assumed to satisfy the usual laws of equality.

Contexts: These are formed by the rules:

$$\frac{}{(\cdot)\ \mathtt{ctx}} \qquad\qquad \frac{\Gamma \vdash A\ \mathtt{type}}{\Gamma, x : A\ \mathtt{ctx}}$$

Here it is assumed that x is a fresh variable, not already occurring in Γ. Note that, unlike in the simple type theory of the previous chapter, the order of the types occurring in a context now matters, since types to the right may depend on ones to their left.

Types: In addition to the usual *simple types*, generated from *basic types* by formation of *products* and *function types*, we may also have some *basic types in context*,

$$\text{Basic dependent types} \quad \Gamma_1 \vdash \mathtt{B}_1,\ \Gamma_2 \vdash \mathtt{B}_2,\ \cdots$$

where the contexts Γ need not be basic. Further dependent types are formed from the basic ones by the *sum* Σ and *product* Π *type formers*, using the *formation rules*:

$$\frac{\Gamma, x : A \vdash B\ \mathtt{type}}{\Gamma \vdash \Sigma_{x:A} B\ \mathtt{type}} \qquad \frac{\Gamma, x : A \vdash B\ \mathtt{type}}{\Gamma \vdash \Pi_{x:A} B\ \mathtt{type}}$$

Terms: As for the simple types, we assume there is a countable set of *variables* $x, y, z, \ldots$. We are also given a set of *basic constants*. The set of *terms* is then generated from variables and basic constants by the following grammar, just as for simple types:

$$\begin{aligned} \text{Variables} \quad & v ::= x \mid y \mid z \mid \cdots \\ \text{Constants} \quad & c ::= \mathtt{c}_1 \mid \mathtt{c}_2 \mid \cdots \\ \text{Terms} \quad & t ::= v \mid c \mid * \mid \langle t_1, t_2 \rangle \mid \mathtt{fst}\, t \mid \mathtt{snd}\, t \mid t_1\, t_2 \mid \lambda x : A \,.\, t \end{aligned}$$

The rules for deriving typing judgments are much as for simple types. They are of course assumed to hold in any context Γ.

- Each basic constant $\mathtt{c}_i$ has a uniquely determined type C_i (not necessarily basic):

$$\frac{}{\mathtt{c}_i : C_i}$$

- The type of a variable is determined by the context:

$$\frac{}{x_1 : A_1, \ldots, x_i : A_i, \ldots, x_n : A_n \vdash x_i : A_i}\ (1 \leq i \leq n)$$

- The constant $*$ has type $\mathtt{1}$:

$$\frac{}{* : \mathtt{1}}$$

- The typing rules for pairs and projections now take the form:

$$\frac{a : A \qquad b : B(a)}{\langle a, b \rangle : \Sigma_{x:A} B} \qquad \frac{c : \Sigma_{x:A} B}{\mathtt{fst}\, c : A} \qquad \frac{c : \Sigma_{x:A} B}{\mathtt{snd}\, c : B(\mathtt{fst}\, c)}$$

  We write e.g. $B(a)$ rather than $B[a/x]$ to indicate a substitution of the term a for the variable x in the type B. Similarly, we may write $\Sigma_{x:A} B(x)$ to emphasize the possible occurence of the variable x in B. We treat $A \times B$ as another way of writing $\Sigma_{x:A} B$, when the variable $x : A$ does not occur in the type B.

- The typing rules for application and λ-abstraction are now:

$$\frac{t : \Pi_{x:A}B \qquad a : A}{t\,a : B(a)} \qquad\qquad \frac{x : A \vdash t : B}{(\lambda x : A \,.\, t) : \Pi_{x:A}B}$$

  We treat $A \to B$ as another way of writing $\Pi_{x:A}B$ when the variable $x : A$ does not occur in the type B.

The (β and η) equations between these terms are just as they were for simple types:

- Equations for unit type:

$$\frac{}{t \equiv * : \mathtt{1}}$$

- Equations for sum types:

$$\frac{u \equiv v : A \qquad s \equiv t : B(a)}{\langle u, s\rangle \equiv \langle v, t\rangle : \Sigma_{x:A}B}$$

$$\frac{s \equiv t : \Sigma_{x:A}B}{\mathtt{fst}\,s \equiv \mathtt{fst}\,t : A} \qquad\qquad \frac{s \equiv t : \Sigma_{x:A}B}{\mathtt{snd}\,s \equiv \mathtt{snd}\,t : A}$$

$$\frac{}{t \equiv \langle \mathtt{fst}\,t, \mathtt{snd}\,t\rangle : \Sigma_{x:A}B} \qquad (\eta\text{-rule})$$

$$\frac{}{\mathtt{fst}\,\langle s, t\rangle \equiv s : A} \qquad\qquad \frac{}{\mathtt{snd}\,\langle s, t\rangle \equiv t : A} \qquad (\beta\text{-rule})$$

- Equations for product types:

$$\frac{s \equiv t : \Pi_{x:A}B \qquad u \equiv v : A}{s\,u \equiv t\,v : B}$$

$$\frac{x : A \vdash t \equiv u : B}{(\lambda x : A \,.\, t) \equiv (\lambda x : A \,.\, u) : \Pi_{x:A}B}$$

$$\frac{}{(\lambda x : A \,.\, t)u \equiv t[u/x] : A} \qquad (\beta\text{-rule})$$

$$\frac{}{\lambda x : A \,.\, (t\,x) \equiv t : \Pi_{x:A}B} \ \text{if } x \notin \mathsf{FV}(t) \qquad (\eta\text{-rule})$$

Equality types: Just as for first-order logic, for each type A we have a primitive *equality type*:

$$x, y : A \vdash \mathtt{Eq}_A(x, y)\ \mathtt{type}\,.$$

This is called *propositional equality*. For convenience, we may sometimes also write $x =_A y$ for $\mathtt{Eq}_A(x, y)$. Although they will turn out to be logically equivalent, the reader is warned not to confuse propositional and judgemental equality $x \equiv y : A$.

The formation, introduction, elimination, and computation rules for equality types are as follows:

$$\frac{s : A \qquad t : A}{s =_A t \ \mathtt{type}} \qquad\qquad \frac{a : A}{\mathtt{refl}_a : (a =_A a)}$$

$$\frac{p : s =_A t}{s \equiv t : A} \qquad\qquad \frac{p : s =_A t}{p \equiv \mathtt{refl}_s : (s =_A s)}$$

The elimination rule is known as *equality reflection*. We may say that two elements $s, t : A$ are *propositionally equal* if the type $s =_A t$ is inhabited. Thus the equality reflection rule says that if two terms are propositionally equal then they are judgementally equal.

Exercise 3.2.1. Show that two terms are propositionally equal if, and only if, they are judgementally equal.

Remark 3.2.2 (Identity types)**.** The formulation of the rules for equality just given is known as the *extensional* theory. There is also an *intensional* version, with different elimination (and computation) rules, to be considered in the next chapter. To help maintain the distinction between these three (!) different relations, the intensional version is sometimes called the *identity type* and written $\mathtt{Id}_A(s, t)$ instead. See [AG] for details.

Remark 3.2.3 (Variant rules for sum types)**.** Another formulation of the rules for Σ-types using a single dependent elimination rule is as follows:

$$\frac{z : \Sigma_{x:A}B \vdash C \ \mathtt{type} \qquad x : A, y : B(x) \vdash c(x, y) : C(\langle x, y\rangle)}{z : \Sigma_{x:A}B \vdash \mathtt{split}(z, c) : \Sigma_{x:A}B}$$

with the associated computation rule:

$$\frac{z : \Sigma_{x:A}B \vdash C \ \mathtt{type} \qquad x : A, y : B(x) \vdash c(x, y) : C(\langle x, y\rangle)}{x : A, y : B(x) \vdash \mathtt{split}(\langle x, y\rangle, c) \equiv c(x, y) : C(\langle x, y\rangle)}$$

These rules permit one to derive the simple elimination terms $\mathtt{fst}\, c$ and $\mathtt{snd}\, c$, and to prove the above computation rules for them. The η-rule is derived using a dependent elimination involving the $\mathtt{Eq}$-type.

Exercise 3.2.4. Prove the simple elimination rules for sum-types (involving $\mathtt{fst}\, c$ and $\mathtt{snd}\, c$) from the dependent ones (involving $\mathtt{split}$).

Remark 3.2.5 (The type-theoretic axiom of choice)**.** One of the oldest problems in the foundations of mathematics is the logical status of the Axiom of Choice. Is it a "Law of Logic"? A mathematical fact about sets? A falsehood with paradoxical consequences?

Per Martin-Löf discovered that the rules of constructive type theory that we have just presented actually suffice to *decide* this question in favor of "Law of Logic" in a certain sense [ML84] (see also [Tai68]). Since the statement of the type theoretic axiom of choice goes (slightly) beyond standard first-order logic, this arguably provides a resolution that also clarifies why the problem remained open for so long in conventional mathematics.

Under propositions as types, reading Σ as "there exists" and Π as "for all", a type such a $\Pi_{x:A}\Sigma_{y:B}R(x,y)$ can be regarded as a stating a proposition—in this case, "for all $x : A$ there is a $y : B$ such that $R(x,y)$". By Curry-Howard, such a "proposition" is then provable if it has a closed term $t : \Pi_{x:A}\Sigma_{y:B}R(x,y)$, which then corresponds to a proof, by unwinding the rules that constructed the term, and observing that they correspond to the usual natural deduction rules for first-order logic.

Of course, the rules of construction for terms correspond to provability only under a certain "constructive" conception of validity (see [Sco70]). Stated as follows,

$$\Pi_{x:A}\Sigma_{y:B}R(x,y) \to \Sigma_{f:A\to B}\Pi_{x:A}R(x,fx)\,, \tag{3.3}$$

the "type theoretic axiom of choice" may sound like the classical axiom of choice under the propositions as types interpretation, but this type is actually *provable* in (constructive) type theory, rather than being an axiom!

Exercise 3.2.6. Prove the type theoretic axiom of choice (3.3) from the rules for sum and product types given here.

3.2.1 Interaction of Eq with Σ and Π

The type theoretic axiom of choice Example 3.2.5, can be seen as a distributivity law for Σ and Π. It is in fact an *isomorphism of types*: there are terms going both ways, the composites of which are propositionally (and therefore definitionally!) equal to the identity maps (i.e. $\lambda x : X.\, x : X \to X$). It is natural to ask, how do the other type formers interact?

Consider first the result of combining Eq-types with Σ. We can show that for $s, t : A \times B$ there is always a term,

$$\mathtt{Eq}_{A\times B}(s,t) \to \big(\mathtt{Eq}_A(\mathtt{fst}\,s, \mathtt{fst}\,t) \times \mathtt{Eq}_B(\mathtt{snd}\,s, \mathtt{snd}\,t)\big)\,,$$

Moreover, there is a term in the other direction as well, and the composites are propositionally equal to the identity. By equality reflection, it therefore follows that these types are also *syntactically isomorphic*, in the sense just described. The same is true for dependent sums, although this is a bit more awkward to state, owing to the fact that $\mathtt{snd}\,(s) : B(\mathtt{fst}\,s)$ and $\mathtt{snd}\,t : B(\mathtt{fst}\,(t)$. However, since the first projection gives a term $p : \mathtt{Eq}_A(\mathtt{fst}\,s, \mathtt{fst}\,t)$ we have $\mathtt{fst}\,s \equiv \mathtt{fst}\,t$ and therefore $B(\mathtt{fst}\,s) \equiv B(\mathtt{fst}\,t)$, so that $\mathtt{Eq}_{B(\mathtt{fst}\,s)}(\mathtt{snd}\,s, \mathtt{snd}\,t)$ makes sense, and is in fact judgementally equal to $\mathtt{Eq}_{B(\mathtt{fst}\,t)}(\mathtt{snd}\,s, \mathtt{snd}\,t)$, so we can write:

$$\mathtt{Eq}_{\Sigma_{x:A}B}(s,t) \to \Sigma_{p:\mathtt{Eq}_A(\mathtt{fst}\,s,\mathtt{fst}\,t)}\mathtt{Eq}_{B(\mathtt{fst}\,s)}(\mathtt{snd}\,s, \mathtt{snd}\,t)\,,$$

Moreover, since $\mathtt{Eq}_{B(\mathtt{fst}\,s)}(\mathtt{snd}\,s, \mathtt{snd}\,t)$ does not depend on $p : \mathtt{Eq}_A(\mathtt{fst}\,s, \mathtt{fst}\,t)$, this actially rewrites to:

$$\mathtt{Eq}_{\Sigma_{x:A}B}(s,t) \to \mathtt{Eq}_A(\mathtt{fst}\,s, \mathtt{fst}\,t) \times \mathtt{Eq}_{B(\mathtt{fst}\,s)}(\mathtt{snd}\,s, \mathtt{snd}\,t)\,.$$

Moreover, these types are also isomorphic.

For Π-types, given terms $f, g : A \to B$, we can form a term of type

$$\mathtt{Eq}_{A\to B}(f,g) \to \Pi_{x:A}\mathtt{Eq}_B(fx, gx)\,.$$

and again, this is an isomorphism of types. The corresponding law for dependent functions $f, g : \Pi_{x:A}B$ takes the more perspicuous form

$$\mathtt{Eq}_{\Pi_{x:A}B}(f,g) \to \Pi_{x:A}\mathtt{Eq}_{B(x)}(fx, gx)\,.$$

And again, this is also an iso. Note that these last two isomorphisms say that two functions are equal just if they are so "pointwise". This principle is called *Function Extensionality.*

Finally, let us consider equality of equality types. Given any terms $a, b : C$ and $p, q : \mathtt{Eq}_C(a,b)$, what more can be said? The principle called *Uniqueness of Identity Proofs* (UIP) asserts that there is always a term of type

$$\mathtt{Eq}_{\mathtt{Eq}_C(a,b)}(p,q)\,.$$

Is there an argument for this principle, analogous to those for the equalities of terms of types Σ and Π? We shall return to this question in the setting of intensional type theory in the next chapter.

Exercise 3.2.7. Prove that extensional type theory satisfies (UIP).

3.3 Locally cartesian closed categories

Recall the following proposition from **??**.

Proposition 3.3.1 (and Definition)**.** *The following conditions on a category $\mathcal{C}$ with a terminal object 1 are equivalent:*

1. *Every slice category $\mathcal{C}/A$ is cartesian closed.*
2. *For every arrow $f : B \to A$ the (post-) composition functor $\Sigma_f : \mathcal{C}/B \to \mathcal{C}/A$ has a right adjoint f^*, which in turn has a right adjoint Π_f.*

$$\begin{tikzcd} B \arrow[r, "f"] & A \end{tikzcd}$$

$$\begin{tikzcd} \mathcal{C}/B \arrow[r, bend left, "\Sigma_f"] \arrow[r, bend right, "\Pi_f"'] & \mathcal{C}/A \arrow[l, "f^*" description] \end{tikzcd}$$

Such a category is called locally cartesian closed*.*

The notation $\Sigma_f \dashv f^* \dashv \Pi_f$ of course anticipates the interpretation of DTT. A common alternate notation is $f_! \dashv f^* \dashv f_*$.

Proof. Suppose every slice of $\mathcal{C}$ is cartesian closed. It suffices to consider the case of (2) with $A = 1$, and to show that the functor $B^* : \mathcal{C} \to \mathcal{C}/_B$ with $B^*(X) = (\pi_2 : X \times B \to B)$ has both left and right adjoints $\Sigma_B \dashv B^* \dashv \Pi_B : \mathcal{C}/_B \to \mathcal{C}$. For Σ_B we can just take the forgetful functor. For $\Pi_B(p : X \to B)$ we use the CCC structure in $\mathcal{C}$ to form the map $p^B : X^B \to B^B$, which we then pull back along the point $'1_B' : 1 \to B^B$ that is the transpose of the identity map $1_B : B \to B$.

$$\begin{tikzcd}
\Pi_B(X) \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & X^B \arrow[d, "p^B"] \\
1 \arrow[r, "'1_B'"'] & B^B
\end{tikzcd}$$

Conversely, if $\mathcal{C}$ is LCC, then in every slice $\mathcal{C}/_X$ we can define the product of $A \to X$ and $B \to X$ as $A \times_X B = \Sigma_A A^* B$ and the exponential as $(B^A)_X = \Pi_A A^* B$. The universal properties are then easily checked. □

Exercise 3.3.2. Verify the details of the proof just sketched for Proposition 3.3.1.

Basic examples of LCCCs

1. We have already seen the hyperdoctrine $\mathcal{C} = \mathsf{Set}$ and $P : \mathsf{Set}^{\mathsf{op}} \to \mathsf{Cat}$ where $P(I) = \mathsf{Set}^I$, with action of $f : J \to I$ on $A : I \to \mathsf{Set}$ by precomposition $f^*A = A \circ f : J \to \mathsf{Set}$, which is strictly functorial. There is an equivalent hyperdoctrine with the slice category $\mathsf{Set}/_I$ as the "category of predicates" and action by pullback $f^* : \mathsf{Set}/_I \to \mathsf{Set}/_J$. The equivalence of categories
$$\mathsf{Set}^I \simeq \mathsf{Set}/_I$$
allows us to use post-composition as the left adjoint $f_! : \mathsf{Set}/_J \to \mathsf{Set}/_I$, rather than the coproduct formula in (3.2). Indeed, this hyperdoctrine structure arises immediately from the locally cartesian closed character of Set. We have the same for any other LCC $\mathcal{E}$, namely the pair $(\mathcal{E}, \mathcal{E}/_{(-)})$ determines a hyperdoctrine, with the action of $\mathcal{E}/_{(-)}$ by pullback, and the left and right adjoints coming from the LCC structure.

2. Another familiar example of a hyperdoctrine arising from LCC structure is presheaves on a small category $\mathbb{C}$, where for the slice category $\widehat{\mathbb{C}}/_X$ we have another category of presheaves, namely
$$\widehat{\mathbb{C}}/_X \cong \widehat{\textstyle\int_{\mathbb{C}} X}\,, \tag{3.4}$$

on the category of elements $\int_{\mathbb{C}} X$. For a natural transformation $f : Y \to X$ we have a functor $\int f : \int Y \to \int X$, which (as usual) induces a triple of adjoints on presheaves,

$$(\textstyle\int f)_! \dashv (\int f)^* \dashv (\int f)_* : \widehat{\int Y} \longrightarrow \widehat{\int X}\,,$$

where the middle functor $(\int f)^*$ is precomposition with $\int f$, which preserves all (co)limits. These satisfy the Beck-Chevalley conditions (up to isomorphism), because this indexed category is equivalent to the one coming from the LCC structure (3.4), which we know satisfies them.

Note that each of the categories $\widehat{\mathbb{C}}/_X$ is therefore also Cartesian closed, so that $\widehat{\mathbb{C}}$ is indeed LCC by Proposition 3.3.1. Moreover each $\widehat{\mathbb{C}}/_X$ also has coproducts $0, X+Y$, so it is a "categorified" Heyting algebra—although we don't make that part of the definition of a hyperdoctrine.

3. An instructive example of a hyperdoctrine that is *not* an LCC is the subcategory of Pos of posets and monotone maps, which we already met in Section **??**, with the "predicates" being the discrete fibrations. For each poset K, let us take as the category of predicates $P(K)$ the full subcategory $\mathsf{dFib}\_K \hookrightarrow \mathsf{Pos}/_K$ consisting of the *discrete fibrations*: monotone maps $p : X \to K$ with the "unique lifting property": for any x and $k \leq p(x)$ there is a unique $x' \leq x$ with $p(x') = k$. Since each category $\mathsf{dFib}/_K$ is equivalent to a category of presheaves $\mathsf{Set}^{K^{\mathsf{op}}}$, and pullback along any monotone $f : J \to K$ preserves discrete fibrations, and moreover commutes with the equivalences to the presheaf categories and the precomposition functor $f^* : \hat{K} \to \hat{J}$, we have a hyperdoctrine if only the Beck-Chevalley conditions hold. We leave this as an exercise for the reader. Finally, observe that dFib cannot be an LCC, simply because it does not have a terminal object; however, every slice of course does one, and so every slice $\mathsf{dFib}/_K$ is a CCC, and therefore also an LCC (since a slice of a slice is a slice).

4. An example formally similar to the foregoing is the non-full subcategory $\mathsf{LocHom} \hookrightarrow \mathsf{Top}$ of topological spaces and local homeomorphisms between them, which also lacks a terminal object, but each slice of which $\mathsf{LocHom}/X \simeq \mathsf{Sh}(X)$ is equivalent to the topos of *sheaves* on the space X, and is therefore CCC (and so LCCC).

5. Fibrations of groupoids. Another, similar, example of a hyperdoctrine not arising simply from an LCCC is the category Grpd of groupoids and homomorphisms, which is not LCC (cf. [Pal03]). We can however take as the category of predicates $P(G)$ the full subcategory $\mathsf{Fib}(G) \hookrightarrow \mathsf{Grpd}/_G$ consisting of the *fibrations* into G: homomorphisms $p : H \to G$ with the "iso lifting property": for any $h \in H$ and $\gamma : g \cong p(h)$ there is some $\vartheta : h' \cong h$ with $p(\vartheta) = \gamma$. Now each category $\mathsf{Fib}(G)$ is biequivalent to a category of presheaves of groupoids $\mathsf{Fib}(G) \simeq \mathsf{Grpd}^{G^{\mathsf{op}}}$. It is not so easy to show that this is a (bicategorical) hyperdoctrine; see [HS98]. This example will be important in the next chapter as a model of *intensional* dependent type theory. The category Cat, with *iso*-fibrations as the "predicates", has a similar character.

Exercise 3.3.3. 1. Verify that the pullback of a discrete fibration $X \to K$ along a monotone map $f : J \to K$ exists in Pos, and is again a discrete fibration.

2. Verify the equivalence of categories $\mathsf{dFib}(K) \simeq \mathsf{Set}^{K^{\mathsf{op}}}$.

3. Show the Beck-Chavelley conditions for the indexed category of discrete fibrations of posets.

Exercise 3.3.4. Let $P : \mathcal{C}^{\mathsf{op}} \to \mathsf{Cat}$ be a hyperdoctrine for which there are equivalences $PC \simeq \mathcal{C}/C$, naturally in C, with respect to the left adjoints $\Sigma_f : \mathcal{C}/A \to \mathcal{C}/B$ for all $f : A \to B$ in $\mathcal{C}$. Show that $\mathcal{C}$ is then LCC.

Exercise 3.3.5. Show that any LCCC $\mathcal{C}$, regarded as a hyperdoctrine, has equality in the sense of Remark 3.1.4.

3.4 Functorial semantics of DTT in LCCCs

We begin by describing a "naive" interpretation of dependent type theory in a locally cartesian closed category which, although not strictly sound, is nonetheless useful and intuitive. In particular, it extends the functorial semantics of simple type theory in CCCs that we developed in the last chapter in a natural way. In a subsequent section, we shall "strictify" the interpretation to one that is fully correct, but technically somewhat more complicated. See Remark 3.4.4 below.

Contexts Γ are interpreted as objects $[\![\Gamma]\!]$, and dependent types $\Gamma \vdash A$ as morphisms into the context $[\![\Gamma]\!]$. To begin, let $\mathcal{C}$ be an LCCC, and interpret the empty context as the terminal object, $[\![\cdot]\!] = 1$. Then to each closed basic type $\cdot \vdash \mathsf{B}$, we assign a type $[\![\mathsf{B}]\!]$ and interpret $[\![\cdot \vdash \mathsf{B}]\!] : [\![\mathsf{B}]\!] \to 1$. Proceeding by recursion, given any type in context $\Gamma \vdash A$, we shall have

$$[\![\Gamma \vdash A]\!] : [\![\Gamma, A]\!] \longrightarrow [\![\Gamma]\!]\,,$$

abbreviating $\Gamma, x : A$ to Γ, A. A basic dependent type $\Gamma \vdash B$ is interpreted by specifying a map $[\![\Gamma \vdash B]\!] : [\![\Gamma, B]\!] \longrightarrow [\![\Gamma]\!]$, where the interpretation $[\![\Gamma]\!]$ is assumed to have been already given. Note that in this way, we also interpret the operation of *context extension*, by taking the domain of the interpretation of a type in context.

Weakening a type in context $\Gamma \vdash C$ to one $\Gamma, A \vdash C$ is interpreted as

$$[\![\Gamma, A \vdash C]\!] = p^*[\![\Gamma \vdash C]\!],$$

that is, the lefthand vertical map in the following pullback square, where the substitution $p : [\![\Gamma, A]\!] \to [\![\Gamma]\!]$ is the canonical projection $p = [\![\Gamma \vdash A]\!]$.

$$\begin{tikzcd}
{[\![\Gamma, A, C]\!]} \arrow[r] \arrow[d, "{[\![\Gamma, A \vdash C]\!]}"'] \arrow[dr, phantom, "\lrcorner", very near start] & {[\![\Gamma, C]\!]} \arrow[d, "{[\![\Gamma \vdash C]\!]}"] \\
{[\![\Gamma, A]\!]} \arrow[r, "p"'] & {[\![\Gamma]\!]}
\end{tikzcd} \tag{3.5}$$

Given $\Gamma, A \vdash B$, we may assume that we already have maps

$$[\![\Gamma, A, B]\!] \xrightarrow{[\![\Gamma, A \vdash B]\!]} [\![\Gamma, A]\!] \xrightarrow{[\![\Gamma \vdash A]\!]} [\![\Gamma]\!] \,,$$

and we use the left and right adjoints to the pullback functor

$$[\![\Gamma \vdash A]\!]^* : \mathcal{C}/_{[\![\Gamma]\!]} \to \mathcal{C}/_{[\![\Gamma, A]\!]}$$

to interpret the eponymous type-forming operations:

$$\begin{aligned} [\![\Gamma \vdash \Sigma_{x:A} B]\!] &= \Sigma_{[\![\Gamma \vdash A]\!]}([\![\Gamma, A \vdash B]\!])\,, \\ [\![\Gamma \vdash \Pi_{x:A} B]\!] &= \Pi_{[\![\Gamma \vdash A]\!]}([\![\Gamma, A \vdash B]\!])\,. \end{aligned}$$

A term $\Gamma \vdash a : A$ is interpreted as a section:

$$\begin{tikzcd} {[\![\Gamma]\!]} \arrow[r, "{[\![\Gamma \vdash a : A]\!]}"] \arrow[dr, "="'] & {[\![\Gamma, A]\!]} \arrow[d, "{[\![\Gamma \vdash A]\!]}"] \\ & {[\![\Gamma]\!]} \end{tikzcd}$$

Finally, as in first-order logic, substitution of a term $\Gamma \vdash a : A$ for a variable $\Gamma, x : A$ in a dependent type $\Gamma, A \vdash B$ is interpreted by taking a pullback,

$$\begin{tikzcd} {[\![\Gamma, B(a)]\!]} \arrow[r] \arrow[d, "{[\![\Gamma \vdash B(a)]\!]}"'] \arrow[dr, phantom, "\lrcorner", very near start] & {[\![\Gamma, A, B]\!]} \arrow[d, "{[\![\Gamma, A \vdash B]\!]}"] \\ {[\![\Gamma]\!]} \arrow[r, "{[\![\Gamma \vdash a : A]\!]}"'] & {[\![\Gamma, A]\!]} \end{tikzcd}$$

and similarly for substitution into terms.

More generally, given any substitution $\gamma : \Delta \to \Gamma$ (a tuple of terms $c_1, \dots, c_n$ in context Δ of types those in $\Gamma = (x_1 : C_1, \dots, x_n : C_n)$), we have a morphism $[\![\gamma]\!] : [\![\Delta]\!] \to [\![\Gamma]\!]$. Then, as in the substitution of a single term $\Gamma \vdash c : C$ for a variable $\Gamma, x : C$, we can obtain a pullback diagram along $[\![\gamma]\!]$:

$$\begin{tikzcd} {[\![\Delta, A(\gamma)]\!]} \arrow[r] \arrow[d, "{[\![\Delta \vdash A(\gamma)]\!]}"'] \arrow[dr, phantom, "\lrcorner", very near start] & {[\![\Gamma, A]\!]} \arrow[d, "{[\![\Gamma \vdash A]\!]}"] \\ {[\![\Delta]\!]} \arrow[r, "{[\![\gamma]\!]}"'] & {[\![\Gamma]\!]} \end{tikzcd}$$

The lefthand vertical map is then by definition the interpretation of the substituted type $\Delta \vdash A(\gamma)$. The interpretation of a substitution into a term $\Delta \vdash a(\gamma) : A(\sigma)$ is similarly induced by pullback.

Finally, we interpret an equality type $x : A, y : A \vdash \mathtt{Eq}_A(x, y)$ as the diagonal of the interpretation of A,

$$\llbracket x, y : A \vdash \mathtt{Eq}_A(x, y) \rrbracket = \Delta_{\llbracket A \rrbracket} : \llbracket A \rrbracket \longrightarrow \llbracket A \rrbracket \times \llbracket A \rrbracket \,.$$

As a map of contexts we then have

$$\llbracket A, A, \mathtt{Eq}_A(x, y) \rrbracket = \llbracket A \rrbracket \longrightarrow \llbracket A \rrbracket \times \llbracket A \rrbracket = \llbracket A, A \rrbracket \,.$$

For $\Gamma, A, A \vdash \mathtt{Eq}_A(x, y)$, with a context Γ, we take the diagonal of the map $\llbracket \Gamma \vdash A \rrbracket : \llbracket \Gamma, A \rrbracket \longrightarrow \llbracket \Gamma \rrbracket$ as an object in the slice category over $\llbracket \Gamma \rrbracket$.

Proposition 3.4.1 ([See84])**.** *The rules of dependent type theory are sound with respect to the naive interpretation in any LCCC,* modulo *the following Remark 3.4.4.*

Proof. One needs to check that all of the typing judgements from Section 3.2 are sound, but much of this work has already been done in the simply typed case, in virtue of Proposition 3.3.1. One thing that we cannot take over from that case is the interpretation of the Eq-types, so let us check those rules by way of example.

The Formation rule (in the empty context) is clearly satified by the stated interpretation:

$$\llbracket x : A, y : A \vdash \mathtt{Eq}_A(x, y) \rrbracket = \Delta_{\llbracket A \rrbracket} : \llbracket A \rrbracket \longrightarrow \llbracket A \rrbracket \times \llbracket A \rrbracket \,. \tag{3.6}$$

For the Introduction rule, we pull back the dependent type $\llbracket x : A, y : A \vdash \mathtt{Eq}_A(x, y) \rrbracket$ along the substitution $\delta : \llbracket z : A \rrbracket \to \llbracket x : A, y : A \rrbracket$ (the diagonal of $\llbracket A \rrbracket$ interpreting the variable contraction $\langle z/x, z/y \rangle$) to obtain the diagram:

$$\begin{tikzcd}
\llbracket z : A, \mathtt{Eq}_A(z, z) \rrbracket \arrow[r] \arrow[d, "\llbracket z : A \vdash \mathtt{Eq}_A(z,z) \rrbracket"'] \arrow[dr, phantom, "\lrcorner", very near start] & \llbracket x : A, y : A, \mathtt{Eq}_A(x, y) \rrbracket \arrow[d, "\llbracket x : A, y : A \vdash \mathtt{Eq}_A(x,y) \rrbracket"] \\
\llbracket z : A \rrbracket \arrow[r, "\delta"'] \arrow[ur, dotted, "\mathtt{refl}_z"] & \llbracket x : A, y : A \rrbracket
\end{tikzcd}$$

The term $\llbracket z : A \vdash \mathtt{refl}_z : \mathtt{Eq}_A(z, z) \rrbracket$ is to be a section of the left hand vertical map, or equivalently, the indicated diagonal. This can be taken to be the identity arrow of $\llbracket A \rrbracket$ by the specification of $\llbracket x : A, y : A \vdash \mathtt{Eq}_A(x, y) \rrbracket$ in (3.6). The Elimination and Computation Rules are left as an exercise. $\square$

Exercise 3.4.2. Complete the verification of the Eq-rules.

Exercise 3.4.3. Verify either the Σ or the Π rules.

Remark 3.4.4 (Coherence)**.** Although it is correct according to the intuition of slice categories in an LCCC, the naive interpretation of substitution as pullback leads to a well-known *coherence problem* [Hof95] for the interpretation of dependent type theory, arising from the fact that pullbacks are only determined up to (canonical) isomorphism. For example, in the foregoing situation, given another substitution $\Phi \vdash \delta : \Delta$, we obtain the two-pullback diagram:

$$\begin{tikzcd}
\llbracket \Phi, A(\gamma)(\delta) \rrbracket \arrow[r] \arrow[d, "\llbracket \Phi \vdash A(\gamma)(\delta) \rrbracket"'] \arrow[dr, phantom, "\lrcorner", very near start] & \llbracket \Delta, A(\gamma) \rrbracket \arrow[r] \arrow[d, "\llbracket \Delta \vdash A(\gamma) \rrbracket"'] \arrow[dr, phantom, "\lrcorner", very near start] & \llbracket \Gamma, A \rrbracket \arrow[d, "\llbracket \Gamma \vdash A \rrbracket"] \\
\llbracket \Phi \rrbracket \arrow[r, "\llbracket \Phi \vdash \delta : \Delta \rrbracket"'] & \llbracket \Delta \rrbracket \arrow[r, "\llbracket \Delta \vdash \gamma : \Gamma \rrbracket"'] & \llbracket \Gamma \rrbracket
\end{tikzcd}$$

However, the composition of substitutions $\llbracket \Delta \vdash \gamma : \Gamma \rrbracket \circ \llbracket \Phi \vdash \delta : \Delta \rrbracket$ across the bottom is the map $\llbracket \Phi \vdash \gamma(\delta) : \Gamma \rrbracket$, and so there is another option for the vertical map on the left, namely the single pullback:

$$\begin{tikzcd}
\llbracket \Phi, A(\gamma(\delta)) \rrbracket \arrow[r] \arrow[d, "\llbracket \Phi \vdash A(\gamma(\delta)) \rrbracket"'] \arrow[dr, phantom, "\lrcorner", very near start] & \llbracket \Gamma, A \rrbracket \arrow[d, "\llbracket \Gamma \vdash A \rrbracket"] \\
\llbracket \Phi \rrbracket \arrow[r, "\llbracket \Phi \vdash \delta(\gamma) : \Gamma \rrbracket"'] & \llbracket \Gamma \rrbracket
\end{tikzcd}$$

Since pullbacks are unique up to iso, there is of course a canonical isomorphism

$$\llbracket \Phi, A(\gamma)(\delta) \rrbracket \;\cong\; \llbracket \Phi, A(\gamma(\delta)) \rrbracket$$

over the base $\llbracket \Phi \rrbracket$, but there is no reason for these two objects (and their associated projections) to be the same. To put the matter succinctly, the action of substitution between contexts on dependent types is *strictly* functorial, but the action of pullback on slice categories is only a *pseudo*functor.

The naive LCCC interpretation, with substitution as pullback, is thus only sound "up to (canonical) isomorphism". The problem becomes more acute in the case of intensional type theory, where the interpretation of certain type formers is not even determined up to isomorphism. We shall consider several solutions to this problem in detail in Section **??** below. For the remainder of this chapter on extensional type theory, however, we can continue to work "up to (canonical) isomorphism" without worrying about coherence.

As was done for simple type theory in Section **??**, we can also again develop the relationship between the type theory and its models using the framework of *functorial semantics.* This is now a common generalization of λ-theories, modeled in CCCs, and first-order logic, modeled in Heyting categories. The first step is to build a syntactic *classifying category* $\mathcal{C}_{\mathbb{T}}$ from a theory $\mathbb{T}$ in dependent type theory, which we then show classifies $\mathbb{T}$-models in LCCCs. We omit the now essentially routine details (given the analogous cases already considered), and merely state the main result, the proof of which is also analogous to the previous cases. A detailed treatment can be found in the seminal paper [See84].

Theorem 3.4.5. *For any theory $\mathbb{T}$ in dependent type theory, the locally cartesian closed syntactic category $\mathcal{C}_{\mathbb{T}}$ classifies $\mathbb{T}$-models, in the sense that for any locally cartesian closed category $\mathcal{C}$ there is an equivalence of categories*

$$\mathsf{Mod}(\mathbb{T},\mathcal{C}) \;\simeq\; \mathsf{LCCC}(\mathcal{C}_{\mathbb{T}},\mathcal{C})\,, \tag{3.7}$$

naturally in $\mathcal{C}$. The morphisms of $\mathbb{T}$-models on the left are the isomorphisms of the underlying structures, and on the right we take the natural isomorphisms of LCCC functors.

As a corollary, again as before, we have that dependent type theory is *complete* with respect to the semantics in locally cartesian closed categories, in virtue of the syntactic construction of the classifying category $\mathcal{C}_{\mathbb{T}}$. Specifically, any theory $\mathbb{T}$ has a canonical interpretation $[-]$ in the syntactic category $\mathcal{C}_{\mathbb{T}}$ which is *logically generic* in the sense that, for any terms $\Gamma \vdash s : A$ and $\Gamma \vdash t : A$, we have

$$\begin{aligned} \mathbb{T} \vdash (\Gamma \vdash u \equiv t : A) &\iff [\Gamma \vdash u : A] = [\Gamma \vdash t : A] \\ &\iff [-] \models (\Gamma \vdash s \equiv t : A)\,. \end{aligned}$$

Thus, for the record, we have:

Proposition 3.4.6. *For any dependently typed theory $\mathbb{T}$,*

$$\mathbb{T} \vdash (\Gamma \vdash u \equiv t : A) \quad \textit{if, and only if,} \quad \mathcal{C}_{\mathbb{T}} \models (\Gamma \vdash u \equiv t : A)\,.$$

Of course, the syntactic model $[-]$ in $\mathcal{C}_{\mathbb{T}}$ is the one associated under (3.7) to the identity functor $\mathcal{C}_{\mathbb{T}} \to \mathcal{C}_{\mathbb{T}}$, *i.e.* it is the *universal* one. It therefore satisfies an equation just in case the equation holds in *all* models, by the classifying property of $\mathcal{C}_{\mathbb{T}}$, and the preservation of satisfaction of equations by LCCC functors (as in Proposition **??**).

Corollary 3.4.7. *For any dependently typed theory $\mathbb{T}$,*

$$\mathbb{T} \vdash (\Gamma \vdash u \equiv t : A) \quad \textit{if, and only if,} \quad M \models (\Gamma \vdash u \equiv t : A) \textit{ for every LCCC model } M.$$

Moreover, a closed type A is inhabited $\vdash a : A$ if, and only if, there is a point $1 \to [\![A]\!]^M$ in every model M.

Remark 3.4.8. In the current setting of *extensional* dependent type theory, soundness and completeness with respect to inhabitation is actually to the same for equations, because $\vdash u \equiv t : A$ just if $\vdash e : \mathtt{Eq}_A(u,t)$ for some (closed) term e, and similarly on the semantic side.

The embedding and completeness theorems of the previous chapter with respect to general presheaf models, Kripke models, and topological and sheaf semantics can also be extended to dependently typed theories. See [AR11, Awo00] for details. The internal language construction and Theorem **??**, associating a dependently typed theory with an LCCC, also extend from simple to dependent type theory. Indeed, this is the main result of [See84]. In view of this result, we hereafter move back and forth freely between syntactic (*i.e.*, type theoretic) and semantic (*i.e.*, categorical) statements and proofs.

Exercise 3.4.9. In the internal logic of an LCCC $\mathcal{E}$, show that the category of types in context $\Gamma \in \mathcal{E}$ is equivalent to the slice category $\mathcal{E}/_{\Gamma}$. (*Hint*: use Eq-types.)

3.5 Inductive types

3.5.1 Sum types

Recall from Chapter **??** the following rules for sum types $0, A+B$ in STT, with term-formers $!\,t$, $\mathtt{inl}\,t$, and $\mathtt{inr}\,t$ and $[x.t_1, x.t_2]u$.

1. The Introduction and Elimination rules are:

$$\frac{\Gamma \mid a : A}{\Gamma \mid \mathtt{inl}\,a : A+B} \qquad \frac{\Gamma \mid b : B}{\Gamma \mid \mathtt{inr}\,b : A+B}$$

$$\frac{\Gamma \mid u : \mathtt{0}}{\Gamma \mid !\,u : C} \qquad \frac{\Gamma, x : A \mid s : C \qquad \Gamma, y : B \mid t : C \qquad \Gamma \mid u : A+B}{\Gamma \mid [x.s, y.t]u : C}$$

2. The Computation rules are the following *equations*.

$$\frac{}{z : C \mid z \equiv !\,u : C} \qquad \frac{}{[x.a, y.b](\mathtt{inl}\,s) \equiv a[s/x] : C} \qquad \frac{}{[x.a, y.b](\mathtt{inr}\,t) \equiv b[t/y] : C}$$

$$\frac{}{u \equiv [x.\mathtt{inl}\,x, y.\mathtt{inr}\,y]u : A+B}$$

$$\frac{}{v\big([x.s, y.t]u\big) \equiv [x.vs, y.vt]u : D}$$

To reformulate these in a form suitable for *dependent* types, we shall change the Elimination rule to allow for a type C *varying over* the type being defined. The idea is that the simple elimination rule for $A+B$ determines a (normal) function of the form $A+B \to C$, while the dependent eliminator determines a "dependent function" $z : A+B \vdash s(z) : C(z)$, i.e. a section s,

$$\begin{tikzcd} C \arrow[d] \\ A+B\,. \arrow[u, bend left, "s"] \end{tikzcd}$$

We designate the former as *recursion* and the latter as *induction*, for reasons which will be become clear shortly, but for now, one can think of $C \to A + B$ as a type-family or "predicate" on $A + B$. Let us consider first the special of the "Boolean truth-values" $\mathtt{Bool} = 1 + 1$.

The Booleans. As a special case of sum types, consider the following rules for the type $\mathtt{Bool} = 1 + 1$, with the obvious renaming of the term-formers (cf. [**?**, Section 5.1])

- Formation rule.

$$\mathtt{Bool}\ \mathtt{type}\,.$$

- Introduction rules.

$$\mathtt{false} : \mathtt{Bool}\,, \qquad \mathtt{true} : \mathtt{Bool}\,.$$

- Elimination rule.

$$\frac{x : \mathtt{Bool} \vdash C(x)\ \mathtt{type} \qquad c_0 : C(\mathtt{false}) \qquad c_1 : C(\mathtt{true})}{x : \mathtt{Bool} \vdash \mathsf{ind}_{\mathtt{Bool}}(x, c_0, c_1) : C(x)}$$

- Computation rules.

$$\mathsf{ind}_{\mathtt{Bool}}(\mathtt{false}, c_0, c_1) \equiv c_0 : C(\mathtt{false})\,, \qquad \mathsf{ind}_{\mathtt{Bool}}(\mathtt{true}, c_0, c_1) \equiv c_1 : C(\mathtt{true}) \qquad (\beta)$$

$$\frac{x : \mathtt{Bool} \vdash c(x) : C(x)\ \mathtt{type}}{x : \mathtt{Bool} \vdash \mathsf{ind}_{\mathtt{Bool}}(x, c(\mathtt{false}), c(\mathtt{true})) \equiv c(x) : C(x)} \qquad (\eta)$$

A basic property of dependent elimination rules is now the following.

Proposition 3.5.1. *The η-rule can be derived from the other rules.*

Proof. We use the dependent $\mathtt{Eq}$-type for $C(x)$ as follows:
By $\mathtt{Eq}$-elim it suffices to show that there is a term

$$x : \mathtt{Bool} \vdash e : \mathsf{Eq}_{C(x)}\big(\mathsf{ind}_{\mathtt{Bool}}(x, c(\mathtt{false}), c(\mathtt{true})), c(x)\big)\,.$$

By $\mathtt{Bool}$-elim it therefore suffices to have terms

$$e_0 : \mathsf{Eq}_{C(\mathtt{false})}\big(\mathsf{ind}_{\mathtt{Bool}}(\mathtt{false}, c(\mathtt{false}), c(\mathtt{true})), c(\mathtt{false})\big)\,,$$
$$e_1 : \mathsf{Eq}_{C(\mathtt{true})}\big(\mathsf{ind}_{\mathtt{Bool}}(\mathtt{true}, c(\mathtt{false}), c(\mathtt{true})), c(\mathtt{true})\big)\,.$$

But by the β-rules, we have

$$\mathsf{ind}_{\mathtt{Bool}}(\mathtt{false}, c(\mathtt{false}), c(\mathtt{true})) \equiv c(\mathtt{false})$$
$$\mathsf{ind}_{\mathtt{Bool}}(\mathtt{true}, c(\mathtt{false}), c(\mathtt{true})) \equiv c(\mathtt{true})\,,$$

and so we can take

$$e_0 := \mathtt{refl} : \mathsf{Eq}_{C(\mathtt{false})}\big(c(\mathtt{false}), c(\mathtt{false})\big)\,,$$
$$e_1 := \mathtt{refl} : \mathsf{Eq}_{C(\mathtt{true})}\big(c(\mathtt{true}), c(\mathtt{true})\big)\,.$$

□

Exercise 3.5.2. Formulate the corresponding dependent rules for sum types $A + B$ and prove the η-rule from the others.

Exercise 3.5.3. Assuming the dependent rules for sum types $A + B$, prove the simple rules, including the η and distributivity computation rules.

3.5.2 Natural numbers

First, recall the following definition, which can be stated in any category with a terminal object. It is usually stated with additional parameters, but this is not required in the case of an LCCC (see [LS88] for a discussion).

Definition 3.5.4 (Lawvere [Law63])**.** A *natural numbers object* in an LCCC $\mathcal{E}$ is an object N equipped with the structure $0 : 1 \to \mathsf{N}$ and $s : \mathsf{N} \to \mathsf{N}$, and initial in the category of such structures.

Spelling this out: given any object $A \in \mathcal{E}$ together with morphisms $a : 1 \to A$ and $f : A \to A$, there is a unique map

$$u : \mathsf{N} \to A\,,$$

making the following diagram in $\mathcal{E}$ commute.

$$\begin{tikzcd}
1 \arrow[r, "a"] \arrow[d, equal] & A \arrow[r, "f"] & A \\
1 \arrow[r, "0"'] & \mathsf{N} \arrow[r, "s"'] \arrow[u, "u"] & \mathsf{N} \arrow[u, "u"']
\end{tikzcd}$$

Of course, this is just a categorical way of stating the usual "definition by recursion" property of the natural numbers. It can be shown to imply the usual Peano axioms in an elementary topos (see [LS88]).

In type theory, we can formulate the corresponding notion as an inductive type. Indeed, the type Nat of natural numbers is the paradigmatic inductive type. The familiar rules for Nat in *simple* type theory are as follows.

- Formation rule.

$$\mathsf{Nat}\ \mathtt{type}$$

- Introduction rules.

$$\mathtt{zero} : \mathsf{Nat} \qquad n : \mathsf{Nat} \vdash \mathtt{succ}(n) : \mathsf{Nat}$$

- Simple elimination rule.

$$\frac{C\ \mathtt{type} \qquad c_0 : C \qquad x : C \vdash c(x) : C}{n : \mathsf{Nat} \vdash \mathtt{rec}(n, c_0, c) : C}$$

Note that in the conclusion we treat $c := \lambda x : C.\, c(x)$ as a term of type $C \to C$.

- Computation rules.

$$\begin{gathered}\mathtt{rec}(\mathtt{zero}, c_0, c) \equiv c_0 : C\\ \mathtt{rec}(\mathtt{succ}(n), c_0, c) \equiv c(\mathtt{rec}(n, c_0, c)) : C\end{gathered}$$

The function $\mathtt{rec}_{(c_0,c)} := \lambda n : \mathsf{Nat}.\, \mathtt{rec}(n, c_0, c) : \mathsf{Nat} \to C$ then satisfies the usual recursion equations:

$$\begin{aligned}\mathtt{rec}_{(c_0,c)}(\mathtt{zero}) &\equiv \mathtt{rec}(\mathtt{zero}, c_0, c) \equiv c_0 : C\,,\\ \mathtt{rec}_{(c_0,c)}(\mathtt{succ}(n)) &\equiv \mathtt{rec}(\mathtt{succ}(n), c_0, c)\\ &\equiv c(\mathtt{rec}(n, c_0, c)\\ &\equiv c(\mathtt{rec}_{(c_0,c)}(n)) : C\,.\end{aligned}$$

Note that this specification follows Definition 3.5.4 of an NNO pretty closely, until we come to the *uniqueness* of the function $\mathtt{rec}_{(c_0,c)} : \mathsf{Nat} \to C$. In order to show that $\mathtt{rec}_{(c_0,c)}$ is the *unique* function satisfying the Computation rules, we would need to add an appropriate η-rule (see Exercise 3.5.9 below). Alternately, we can strengthen the elimination rule to a dependent one, in order to allow the type $\mathtt{Eq}_{\mathsf{Nat}}$ to appear in the conclusion, as we did for the example of `Bool` in Proposition 3.5.1. Such a dependent elimination rule corresponds (under propositions as types) to the familiar rule of "proof by induction": if for some property of natural numbers $P(n)$, we have $P(0)$, and if $P(n)$ implies $P(n+1)$ for all n, then $P(n)$ holds for all n. Reformulating this familiar principle in *dependent* type theory with an explicit proof term for the inference from n to $n+1$ results in a more powerful recursion schema with parameters.

Dependent elimination. The rules for Nat in dependent type theory use the same formation and introduction rules as above, but provide for eliminating into a type *family*, parametrized by natural numbers.

- Dependent elimination rule.

$$\frac{n : \mathsf{Nat} \vdash C(n)\ \mathtt{type} \qquad c_0 : C(\mathtt{zero}) \qquad n : \mathsf{Nat}, x : C(n) \vdash c(n, x) : C(\mathtt{succ}(n))}{n : \mathsf{Nat} \vdash \mathtt{ind}(n, c_0, c) : C(n)}$$

  Note that in the conclusion we now treat $c := \lambda n : \mathsf{Nat}\, \lambda x : C(n).\, c(n, x)$ as a term of type $\Pi_{n:\mathsf{Nat}}.\, C(n) \to C(\mathtt{succ}(n))$.

- Dependent computation rules.

$$\begin{aligned}\mathtt{ind}(\mathtt{zero}, c_0, c) &\equiv c_0 : C(\mathtt{zero})\,,\\ \mathtt{ind}(\mathtt{succ}(n), c_0, c) &\equiv c(n, \mathtt{ind}(n, c_0, c)) : C(\mathtt{succ}(n))\,.\end{aligned}$$

Proposition 3.5.5. *In the classifying category $\mathcal{C}_{\mathsf{Nat}}$ with the dependent elimination and computation rules just stated, the type* Nat *equipped with* $\mathtt{zero} : \mathsf{Nat}$ *and* $\mathtt{succ} : \mathsf{Nat} \to \mathsf{Nat}$ *is a natural numbers object.*

Proof. Let A be any type with a distinguished point $a : A$ and an endo map $f : A \to A$. We need to find a unique map

$$u : \mathsf{Nat} \to A\,,$$

making the following diagram commute in $\mathcal{C}_{\mathsf{Nat}}$.

$$\begin{tikzcd}
1 \arrow[r, "a"] \arrow[d, equal] & A \arrow[r, "f"] & A \\
1 \arrow[r, "\texttt{zero}"'] & \mathsf{Nat} \arrow[r, "\texttt{succ}"'] \arrow[u, "u"] & \mathsf{Nat} \arrow[u, "u"']
\end{tikzcd}$$

We therefore require a term $n : \mathsf{Nat} \vdash u(n) : A$ with:

$$\begin{gathered} u(\texttt{zero}) \equiv a : A\,, \\ n : \mathsf{Nat} \vdash u(\texttt{succ}(n)) \equiv f(u(n)) : A\,. \end{gathered} \tag{3.8}$$

In the (dependent) elimination rule, we can take $C(n)$ to be A (the constant family), and c_0 to be a, and for $n : \mathsf{Nat}, x : A$ we let $c(n, x)$ be $f(x)$. In the conclusion we then obtain $n : \mathsf{Nat} \vdash \texttt{ind}(n, a, f) : A$, which we take to be the required map $u : \mathsf{Nat} \to A$. The computation rules then clearly provide the required equations (3.8).

We then use the equality type to prove uniqueness. Namely, suppose that we also have $n : \mathsf{Nat} \vdash v(n) : A$ with

$$\begin{gathered} v(\texttt{zero}) \equiv a : A\,, \\ n : \mathsf{Nat} \vdash v(\texttt{succ}(n)) \equiv f(v(n)) : A\,. \end{gathered} \tag{3.9}$$

We wish to show $u \equiv v : \mathsf{Nat} \to \mathsf{Nat}$. By equality reflection, it suffices to show $\mathsf{Eq}_{\mathsf{Nat}\to\mathsf{Nat}}(u, v)$, and by function extensionality (3.2.1), it is therefore enough to show $\Pi_{n:\mathsf{Nat}}.\mathsf{Eq}_{\mathsf{Nat}}(u(n), v(n))$. Thus we will be done by the following dependent elimination with $C(n) = \mathsf{Eq}_{\mathsf{Nat}}(u(n), v(n))$, once we have found suitable terms e_0 and $e(n, x)$.

$$\frac{\begin{array}{l} e_0 : \mathsf{Eq}_{\mathsf{Nat}}(u(\texttt{zero}), v(\texttt{zero})) \\ n : \mathsf{Nat}, x : \mathsf{Eq}_{\mathsf{Nat}}(u(n), v(n)) \vdash e(n, x) : \mathsf{Eq}_{\mathsf{Nat}}\big(u(\texttt{succ}(n)), v(\texttt{succ}(n))\big) \end{array}}{n : \mathsf{Nat} \vdash \texttt{ind}(n, e_0, e) : \mathsf{Eq}_{\mathsf{Nat}}(u(n), v(n))}$$

But since $u(\texttt{zero}) \equiv a \equiv v(\texttt{zero})$ by (3.8) and (3.9), we can take $e_0 := \texttt{refl}$. By the same, we also have $u(\texttt{succ}(n)) \equiv f(u(n))$ and $v(\texttt{succ}(n)) \equiv f(v(n))$, and by the assumption $x : \mathsf{Eq}_{\mathsf{Nat}}(u(n), v(n))$ we have $u(n) \equiv v(n)$ and thus $f(u(n)) \equiv f(v(n))$. So for $e(n, x)$ we can again take a suitable $\texttt{refl}$. □

Conversely, we also have the following result:

Proposition 3.5.6. *Let $\mathcal{E}$ be an LCCC with a natural numbers object,*

$$1 \xrightarrow{0} \mathsf{N} \xrightarrow{s} \mathsf{N}\,.$$

Then $(\mathsf{N}, 0, s)$ satisfies the Formation, Introduction, (dependent) Elimination, and (dependent) Computation rules for the type Nat*.*

Proof. The Formation and Introduction rules are immediate. For the Elimination rule, suppose given an interpretation of the premises, namely: a dependent type $[\![\mathsf{N} \vdash C]\!] : C \to \mathsf{N}$, a section $c_0 : 1 \to 0^*C$ of the pullback of C over $0 : 1 \to \mathsf{N}$, and a map c as in the following diagram:

$$\begin{tikzcd}
C \arrow[r, "c"] \arrow[dr] & s^*C \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & C \arrow[d] \\
 & \mathsf{N} \arrow[r, "s"'] & \mathsf{N}
\end{tikzcd}$$

Then we have, equivalently, a commutative diagram:

$$\begin{tikzcd}
1 \arrow[r, "c_0"] \arrow[d, equal] & C \arrow[r, "c"] \arrow[d] & C \arrow[d] \\
1 \arrow[r, "0"'] & \mathsf{N} \arrow[r, "s"'] & \mathsf{N}
\end{tikzcd}$$

By the universal property of N as an NNO, there is a (unique!) section $i : \mathsf{N} \to C$ commuting with the NNO structure maps.

$$\begin{tikzcd}
1 \arrow[r, "c_0"] \arrow[d, equal] & C \arrow[r, "c"] \arrow[d] & C \arrow[d] \\
1 \arrow[r, "0"'] & \mathsf{N} \arrow[r, "s"'] \arrow[u, bend left, "i"] & \mathsf{N} \arrow[u, bend right, "i"']
\end{tikzcd}$$

Taking $[\![n : \mathsf{Nat} \vdash \mathtt{ind}(n, c_0, c)]\!] := i$ then satisfies the dependent Elimination and Computation rules. (Why is $i : \mathsf{N} \to C$ a *section* of $C \to \mathsf{N}$?) ☐

Remark 3.5.7. We shall see that, in dependent type theory, the foregoing two propositions are typical of inductive types in general: a structured type (S, s) is *initial* if and only if every type family $T \to S$ equipped with the same kind of structure (T, t) over (S, s) has a structure preserving section. We shall make this precise in terms of *algebras for (polynomial) endofunctors* in the next section.

Exercise 3.5.8. Use the dependent rules for Nat to define the addition function $+ : \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat}$ in such a way that

$$m + 0 \equiv m\,,$$
$$m + \mathtt{succ}(n) \equiv \mathtt{succ}(m + n)\,.$$

Exercise 3.5.9. Formulate a (simple) η-rule for Nat that allows one to prove the analog of Proposition 3.5.5 from just the simple elimination and computation rules (including your new η-rule).

3.5.3 Algebras for endofunctors

Let $\mathcal{E}$ be an LCCC with sums, and consider the endofunctor $F : \mathcal{E} \to \mathcal{E}$ with $F(X) = 1+X$. As usual, an *algebra* for F is an object A equipped with a map $a : F(A) \to A$, and a homomorphism of F-algebras $h : (A, a) \to (B, b)$ is a map $h : A \to B$ commuting with the algebra structure maps:

$$\begin{tikzcd} FA \arrow[r, "Fh"] \arrow[d, "a"'] & FB \arrow[d, "b"] \\ A \arrow[r, "h"'] & B \end{tikzcd} \tag{3.10}$$

In this particular case, such an algebra $a : FA = 1 + A \to A$ corresponds to a unique "successor algebra" structure $a = [a_0, a_s]$ where:

$$1 \xrightarrow{a_0} A \xrightarrow{a_s} A\,,$$

and an F-algebra homomorphism is just a successor algebra homomorphism. It follows that a *natural numbers object* is the same thing as an *initial F-algebra*, i.e. an initial object in the category $F\text{-}\mathsf{Alg}$ of F-algebras and their homomorphisms.

More generally, one can consider the category of algebras for *any* endofunctor $F : \mathcal{E} \to \mathcal{E}$, but there need not always be an initial one, in light of the following fact.

Lemma 3.5.10 (Lambek). *Given $F : \mathcal{E} \to \mathcal{E}$, if $i : F(I) \to I$ is an initial F-algebra, then the map i is an isomorphism.*

Exercise 3.5.11. Prove Lambek's lemma and conclude that not every endofunctor has an initial algebra.

When an initial algebra does exists, it can be regarded as a generalized "inductive type", in view of the following.

Proposition 3.5.12. *Let $F : \mathcal{E} \to \mathcal{E}$ and let $i : F(I) \to I$ be an initial F-algebra. Let $p : C \to I$ be a family over I with an F-algebra structure $c : FC \to C$ making the following diagram commute.*

$$\begin{tikzcd} FC \arrow[r, "c"] \arrow[d, "Fp"'] & C \arrow[d, "p"] \\ FI \arrow[r, "i"'] & I \end{tikzcd}$$

Then there is a section $s : I \to C$ that is an algebra homomorphism.

Conversely, if $a : FA \to A$ is an algebra such that every algebra $(C, c) \to (A, a)$ over it has an algebra section, then (A, a) is initial in the category $F\text{-}\mathsf{Alg}$.

Proof. Since (I, i) is initial, there is an algebra homomorphism $h : (I, i) \to (C, c)$. Since homomorphisms compose, $p \circ h : I \to I$ is also one. But then $p \circ h = 1_I$ since homomorphisms from initial algebras are unique, and (I, i) is initial.

Conversely, suppose every algebra over (A, a) has an (algebra) section, and let (B, b) be any algebra. We need a (unique) algebra homomorphism $u : (A, a) \to (B, b)$. Consider the projection $p_1 : A \times B \to A$ as an algebra over (A, a) with structure map $(aFp_1,\, bFp_2) : F(A \times B) \to A \times B$.

$$\begin{tikzcd}
F(A \times B) \arrow[r, "{(aFp_1, bFp_2)}"] \arrow[d, "Fp_1"'] & A \times B \arrow[d, "p_1"] \\
FA \arrow[r, "a"'] & A
\end{tikzcd}$$

Let $s : A \to A \times B$ be an algebra section, so that $u := p_2 \circ s : A \to B$ is a homomorphism. We claim that u is unique. Indeed, given any homomorphism $t : A \to B$, consider the equalizer $e : E(t, u) \rightarrowtail A$, which of course is the pullback of the diagonal $\Delta_B : B \rightarrowtail B \times B$ along the map $(t, u) : A \to B \times B$. It will suffice to equip $E(t, u)$ with an algebra structure map $\varepsilon : F(E(t, u)) \to E(t, u)$ over $a : FA \to A$, for then we shall have an algebra section $s : A \to E(t, u)$, whence $e : E(t, u) \cong A$, and so $t = u$, since $e : E(t, u) \rightarrowtail A$ is the equalizer.

$$\begin{tikzcd}
F(E(t,u)) \arrow[r, dotted, "\varepsilon"] \arrow[d, "Fe"'] & E(t,u) \arrow[d, tail, bend left, "e"] \\
FA \arrow[r, "a"'] & A \arrow[u, tail, bend left, "s"]
\end{tikzcd}$$

We claim that $a \circ Fe : F(E(t, u)) \to FA \to A$ precomposes equally with $t, u : A \to B$, which will suffice for ε. But this follows by a chase around the following diagram, recalling that t and u are homomorphisms, and $te = ue$.

$$\begin{tikzcd}
F(E(t,u)) \arrow[r, dotted, "\varepsilon"] \arrow[d, "Fe"'] & E(t,u) \arrow[d, tail, "e"] \\
FA \arrow[r, "a"'] \arrow[d, shift right, "Ft"'] \arrow[d, shift left, "Fu"] & A \arrow[d, shift right, "t"'] \arrow[d, shift left, "u"] \\
FB \arrow[r, "b"'] & B
\end{tikzcd}$$

□

Exercise 3.5.13. Reformulate and prove Proposition 3.5.12 in dependent type theory, using `Eq`-types for the equalizer.

Polynomial endofunctors. One class of endofunctors $F : \mathsf{Set} \to \mathsf{Set}$ for which initial algebras *do* exist are the (finitary) *polynomial functors*,

$$F(X) = C_0 + C_1 \times X + \cdots + C_n \times X^n\,, \tag{3.11}$$

where $C_0, \ldots, C_n$ are sets and $X^k = X \times \ldots \times X$ is the k-fold product. The endofunctor $1 + X$ for the natural numbers $\mathbb{N}$ was of course of this kind. An algebra (A, a) for e.g. the functor $1 + X + X^2$ will be a *pregroup structure* on the set A,

$$[a_0, a_1, a_2] : 1 + A + A^2 \longrightarrow A\,,$$

corresponding to an element $a_0 \in A$, a unary operation $a_1 : A \to A$ and a binary operation $a_2 : A \times A \to A$. We say "pregroup structure" to emphasize that no equations are required to hold; this is just an interpretation of the "signature" of a group.

An algebra $a : F(A) \to A$ for the functor (3.11) would thus be a "conventional" algebraic structure on A (in the sense of universal algebra) consisting of C_0-many "constants" $1 \to A$ and C_1-many "unary operations" $A \to A$, ..., and C_n-many n-ary "operations" $A^n \to A$. Note that algebra homomorphisms in the sense of (3.10) are just homomorphisms in the usual sense of algebraic structures.

Proposition 3.5.14. *Any finitary polynomial functor* $F : \mathsf{Set} \to \mathsf{Set}$ *such as* (3.11) *has an initial algebra.*

An elementary proof would proceed by forming the "term algebra" A consisting of all expressions of the form $a_{c_k}(t_1, \ldots, t_k)$, where $c_k \in C_k$, and the t_k are "previously" formed terms of the same kind. A more abstract proof (that also generalizes to other settings) is as follows:

Proof. By [Awo10, 10.13], it suffices to show that F preserves ω-colimits, for then the colimit of the sequence

$$0 \to F0 \to FF0 \to \ldots$$

will be an initial algebra. The coproduct $C_0 + C_1 \times X + \cdots + C_n \times X^n$ preserves all of the colimits preserved by the monomials $C_k \times X^k$, and each of these preserves the colimits preserved by the functor $X^k = X \times \ldots \times X$, which includes the filtered ones like ω. □

The proof obviously generalizes to a much larger class of endofunctors, including ones on categories other than Set (see e.g. [Awo10, 10.14]). Rather than pursuing this topic further, however (for which, see [AR94]), we want to consider a type-theoretic reformulation that captures a range of inductive types with good properties. Let us first apply Proposition 3.5.12, and state the resulting rules for an initial algebra of a polynomial functor, for example:

$$F(X) = A + B \times X + C \times X^2\,,$$

as a simplified version of (3.11) (the general case will be covered below). Let us write $s : F(I) \to I$ for an initial F-algebra (assuming it exists). Then we have the following rules:

The assumption that the initial algebra exists takes the form:

- I-formation rule.

$$\frac{A\ \mathtt{type} \qquad B\ \mathtt{type} \qquad C\ \mathtt{type}}{I\ \mathtt{type}}$$

And I also will come with an algebra structure $s : F(I) \to I$, given by:

- I-introduction rules for the operation $s : A + B{\times}I + C{\times}I^2 \longrightarrow I$,

$$\frac{a : A}{s_0(a) : I} \qquad \frac{b : B\,,i : I}{s_1(b,i) : I} \qquad \frac{c : C\,,i : I\,,j : I}{s_2(c,i,j) : I}$$

We also have an induction principle, as in Proposition 3.5.12, for any F-algebra $f : F(X) \to X$ over $F(I) \to I$:

- I-elimination rule.

$$\frac{\begin{array}{l} i : I \vdash X(i)\ \mathtt{type} \\ a : A\ \vdash f_0(a) : X(s_0(a)) \\ b : B,\ i : I,\ x : X(i)\ \vdash f_1(b,x) : X(s_1(b,i)) \\ c : C,\ i,j : I,\ x : X(i),\ y : X(j)\ \vdash\ f_2(c,x,y) : X(s_2(c,i,j)) \end{array}}{i : I \vdash \mathtt{rec}(i, f_0, f_1, f_2) : X(i)}$$

And, of course, there is a computation rule, resulting from first introducing and then eliminating, which says that the following diagram commutes.

$$\begin{tikzcd} F(X) \arrow[r, "f"] \arrow[d] & X \arrow[d] \\ FI \arrow[r, "s"'] \arrow[u, bend left, "\mathtt{rec}"] & I \arrow[u, bend right, "\mathtt{rec}"'] \end{tikzcd}$$

- I-computation rules.

$$\begin{aligned} a : A\ &\vdash \mathtt{rec}(s_0(a), f_0, f_1, f_2)\ \equiv\ f_0(a) : X(s(a))\,, \\ b : B,\ i : I\ &\vdash \mathtt{rec}(s_1(b,i), f_0, f_1, f_2)\ \equiv\ f_1(b, \mathtt{rec}(i, \vec{f})) : X(s_1(b,i))\,, \\ c : C,\ i,j : I\ &\vdash \mathtt{rec}(s_2(c,i,j), f_0, f_1, f_2)\ \equiv\ f_2(c, \mathtt{rec}(i, \vec{f}), \mathtt{rec}(j, \vec{f})) : X(s_2(c,i,j))\,. \end{aligned}$$

Exercise 3.5.15. Specialize the foregoing to the case $F(X) = 1 + X$ and derive the rules for Nat from Section 3.5.2.

3.5.4 W-types

The rules just given for initial algebras for polynomial functors are a bit unwieldy as the degree of the polynomial F increases, but they simplify when stated in a more general form, which can actually be applied in any LCCC $\mathcal{E}$. Indeed, let $p : B \to A$ be any map in $\mathcal{E}$, regarded as a type family $a : A \vdash B(a)$. We can form the (generalized) polynomial endofunctor $P : \mathcal{E} \to \mathcal{E}$ as:

$$P(X) = \Sigma_{a:A} X^{B(a)}\ =\ A_! \circ p_* \circ B^*(X)\,, \tag{3.12}$$

as indicated in the following:

$$\begin{tikzcd}
X & X \times B \arrow[l] \arrow[d] & P(X) \arrow[d] \\
 & B \arrow[r, "p"'] & A
\end{tikzcd}$$

Observe that $B^* = p^* \circ A^*$, so that

$$\Sigma_A \, p_* \, B^* \, (X) = \Sigma_A \, p_* \, p^* A^*(X) = \Sigma_A (A^* X)^p \, ,$$

which justifies the polynomial notation $\Sigma_{a:A} X^{B(a)}$, since the type $B(a)$ is the fiber of $p : B \to A$ at $a : A$.

Definition 3.5.16 (cf. [MP00])**.** A (semantic) W-*type* in a locally cartesian closed category $\mathcal{E}$ is an initial algebra for a polynomial endofunctor $P : \mathcal{E} \to \mathcal{E}$ associated to a map $p : B \to A$, as in (3.12).

We shall see that W-types can be used to introduce a wide class of inductive types in dependent type theory. The following rules for W-types are due to [**?**]. To state them more perspicuously, for a fixed type family $x : A \vdash B(x)$ we may write W instead of $\mathsf{W}_{x:A} B(x)$.

- W-formation rule.
$$\frac{A \ \mathtt{type} \qquad x : A \vdash B(x) \ \mathtt{type}}{\mathsf{W}_{x:A} B(x) \ \mathtt{type}}$$

- W-introduction rule.
$$\frac{a : A \qquad t : B(a) \to \mathsf{W}}{\mathtt{wsup}(a, t) : \mathsf{W}}$$

- W-elimination rule.
$$\frac{\begin{array}{l} w : \mathsf{W} \vdash C(w) \ \mathtt{type} \\ x : A, \ u : B(x) \to \mathsf{W}, \ v : \Pi_{y:B(x)} C(u(y)) \ \vdash \ c(x, u, v) : C(\mathtt{wsup}(x, u)) \end{array}}{w : \mathsf{W} \vdash \mathtt{wrec}(w, c) : C(w)}$$

- W-computation rule.
$$\frac{\begin{array}{l} w : \mathsf{W} \vdash C(w) \ \mathtt{type} \\ x : A, \ u : B(x) \to \mathsf{W}, \ v : \Pi_{y:B(x)} C(u(y)) \ \vdash \ c(x, u, v) : C(\mathtt{wsup}(x, u)) \end{array}}{\begin{array}{r} x : A, \ u : B(x) \to \mathsf{W} \vdash \mathtt{wrec}(\mathtt{wsup}(x, u), c) \ \equiv \\ c(x, u, \lambda y. \mathtt{wrec}(u(y), c)) : C(\mathtt{wsup}(x, u)) \end{array}}$$

Informally, the W-type for a family $x : A \vdash B(x)$ can be regarded as the free algebra for a signature with A-many operations, each of (possibly infinite) arity $B(a)$ – and *no* equations. Indeed, the premisses of the formation rule above can be thought of as specifying a signature that has the terms $a : A$ as the operations themselves, and (the cardinality of) the type $B(a)$ as the "arity" of $a : A$. Then, the introduction rule specifies an element of the free algebra, namely $\mathtt{wsup}(a, t) : \mathsf{W}$, where $t : \mathsf{W}^{B(a)}$. The elimination rule then states that W is the *initial algebra* of the assocated polynomial functor $\Sigma_{a:A} X^{B(a)}$.

Proposition 3.5.17. *An object* W *satisfies the rules for* W*-types if, and only if, it is an initial algebra for the polynomial functor* $P(X) = \Sigma_{a:A} X^{B(a)}$.

Proof. We use Proposition 3.5.12. Suppose W satisfies the rules for W types above. From the introduction rule, we have a P-algebra structure $\mathtt{wsup} : P(\mathsf{W}) \to \mathsf{W}$. Let $C \to \mathsf{W}$ with a P-algebra structure $c : P(C) \to C$ over $\mathtt{wsup} : P(\mathsf{W}) \to \mathsf{W}$. This is exactly what the premises of the elimination rule say, so by the conclusion of that rule there is a section $w : \mathsf{W} \vdash \mathtt{wrec}(w, c) : C(w)$, which is a P-algebra homomorphism by the computation rule. The converse is left as an exercise. □

Exercise 3.5.18. Complete the proof of Proposition 3.5.17.

Remark 3.5.19 (cf. [AGS17])**.** The foregoing (dependent) W-elimination rule implies what may be called the *simple* W-elimination rule:

$$\frac{C\ \mathtt{type} \qquad x : A, v : B(x) \to C \vdash c(x, v) : C}{w : \mathsf{W} \vdash \mathsf{simp\text{-}wrec}(w, c) : C}$$

This can be recognized as a recursion principle for maps from W into P-algebras, since the premisses of the rule describe exactly a type C equipped with a structure map $c : PC \to C$. For this special case of the elimination rule, the corresponding computation rule again states that the function

$$\lambda w.\mathsf{simp\text{-}wrec}(w, c) : \mathsf{W} \to C\,,$$

where $c(x, v) = c(\langle x, v\rangle)$ for $x : A$ and $v : B(x) \to C$, is a P-algebra homomorphism. Moreover, this homomorphism can then be shown to be (definitionally) *unique*, using $\mathtt{Eq}$-types, the elimination rule, and the reflection rule, as in the proof of Proposition 3.5.12. The converse implication also holds: one can derive the general W-elimination rule from the simple elimination rule and the following η-rule.

$$\frac{\begin{array}{l} C : \mathtt{type} \qquad w : \mathsf{W} \vdash h(w) : C \\ x : A, v : B(x) \to C \vdash c(x, v) : C \\ x : A\,, u : B(x) \to W \vdash h\,(\mathtt{wsup}(x, u)) = c(x, \lambda y.hu(y)) : C \end{array}}{w : \mathsf{W} \vdash h(w) \equiv \mathsf{simp\text{-}wrec}(w, c) : C}$$

This rule states the uniqueness of the $\mathsf{simp\text{-}wrec}$ term among algebra maps. Overall, we therefore have that induction and recursion are inter-derivable in the present theory with extensional Eq-types:

Induction	$\Leftrightarrow$	**Recursion**
Dependent elimination		Simple elimination
Dependent computation		Simple computation + η-rule

Examples of W-types. We conclude by noting that many familiar inductive types can be reduced to W-types. We mention the following examples, among many others (see [?], [?], [?], [?], [?], [?]):

1. *Natural numbers.* The usual rules for Nat as an inductive type can be derived from its formalization as a W-type. Consider the signature determined by the map `inl` : $1 \to 1+1$ (say): it has two operations, one of which has arity 0 and one of which has arity 1, since these are the pullbacks of the map `inl` $: 1 \to 1+1$ along the two points $1 \rightrightarrows 1+1$. To present this in type theory, we need a type family over `Bool` (say) with the types 0 and 1 as its values. Consider the family

$$v : \mathtt{Bool} \vdash \mathtt{Eq}_{\mathtt{Bool}}(v, \mathtt{true}) \;\; \mathtt{type},$$

which has the values:

$$\mathtt{Eq}_{\mathtt{Bool}}(v, \mathtt{true}) \cong \begin{cases} 1 & v = \mathtt{true} \\ 0 & v = \mathtt{false} \end{cases}$$

Indeed, one can show that the projection $\Sigma_{v:\mathtt{Bool}}\mathtt{Eq}_{\mathtt{Bool}}(v, \mathtt{true}) \to \mathtt{Bool}$ is isomorphic to the map $\mathtt{true} : 1 \to \mathtt{Bool}$ over `Bool`.

The corresponding polynomial functor can then be defined as

$$\begin{aligned} P(X) &= \Sigma_{v:\mathtt{Bool}}\, \mathtt{Eq}_{\mathtt{Bool}}(v, \mathtt{true}) \to X \\ &= 1 + X\,. \end{aligned}$$

The corresponding W type is then the initial algebra of $P(X) = 1 + X$, namely the type Nat of natural numbers,

$$\mathsf{Nat} = \mathsf{W}_{v:\mathtt{Bool}}\, \mathtt{Eq}_{\mathtt{Bool}}(v, \mathtt{true})\,.$$

The canonical element $\mathtt{zero} : \mathsf{Nat}$ and the successor function $\mathtt{succ} : \mathsf{Nat} \to \mathsf{Nat}$ result from the two cases of introduction rule,

$$\frac{b : \mathtt{Bool} \qquad t : \mathtt{Eq}_{\mathtt{Bool}}(b, \mathtt{true}) \to \mathsf{Nat}}{\mathtt{wsup}(b, t) : \mathsf{Nat}}$$

namely:

$$\frac{\mathtt{false} : \mathtt{Bool} \qquad t : (\mathtt{Eq}_{\mathtt{Bool}}(\mathtt{false}, \mathtt{true}) \to \mathsf{Nat})}{\mathtt{wsup}(\mathtt{false}, t) : \mathsf{Nat}}$$

and

$$\frac{\texttt{true} : \texttt{Bool} \qquad t : (\mathtt{Eq}_{\texttt{Bool}}(\texttt{true}, \texttt{true}) \to \mathsf{Nat})}{\texttt{wsup}(\texttt{true}, t) : \mathsf{Nat}}$$

Thus we can take

$$\begin{aligned} \texttt{zero} &:= \lambda t.\, \texttt{wsup}(\texttt{false}, t) : 1 \to \mathsf{Nat}\,, \\ \texttt{succ} &:= \lambda t.\, \texttt{wsup}(\texttt{true}, t) : \mathsf{Nat} \to \mathsf{Nat}\,. \end{aligned}$$

2. *Second number class.* As shown in [?], the second number class can be obtained as a W-type determined by the polynomial functor

$$P(X) = 1 + X + (\mathsf{Nat} \to X)\,.$$

This has algebras with three operations, one of arity 0, one of arity 1, and one of arity (the cardinality of) Nat.

3. *Lists.* The type $\mathsf{List}(A)$ of finite lists of elements of some type A can be built as a W-type determined by the polynomial functor

$$P(X) = 1 + A \times X\,,$$

associated to the map $! + A : 0 + A \to 1 + 1$. We refer to [?] for details.

Exercise 3.5.20. If a signature for an algebraic theory has no constants, then the free algebra on the empty set 0 will itself be empty, as can be seen by considering the term algebra construction of the free algebra $F(0)$. Something similar is true for W-types (say, in Set): if $p : B \twoheadrightarrow A$ is an epimorphism, then $P(0) = \Sigma_{a:A}\, 0^{B(a)} \cong 0$, and so $\mathsf{W}_{a:A}\, B(a) \cong 0$. Prove this.

3.6 Dependent type theory with FOL

Even under the Propositions-as-Types (PaT) conception there are certain types P that are *proof irrelevant* in the sense that for any $p, q : P$, we have $\mathtt{Eq}_P(p, q)$ (meaning that we have a term $t : \mathtt{Eq}_P(p, q)$, and so $p \equiv q$). For example, the type 1 has this property, as does 0. Let us call this such special types *propositional*, which is *definable* by

$$\mathsf{IsProp}(P) \;=\; \Pi_{p,q:P} \mathtt{Eq}_P(p, q)\,.$$

This condition is equivalent to $P \cong P \times P$.

Exercise 3.6.1. Prove this: a type P is propositional if and only if $P \cong P \times P$ (canonically) and, moreover, if and only if the unique map $P \to 1$ is a monomorphism.

The propositions are easily seen to be closed under finite products $P \times Q$, and if $x : X \vdash P(x)$ is a family of propositions, then $\Pi_{x:X} A(x)$ is also a proposition. Finally, if P is a proposition, then so is $A \to P$ for any A.

Exercise 3.6.2. Prove the last three statements.

It therefore makes sense to expect that for any type A, there could be a universal propositional approximation $p : A \to P_A$, with the universal property that every map $A \to P$ with P a proposition factors (uniquely) through $p : A \to P_A$, as in:

$$\begin{tikzcd}
A \arrow[r] \arrow[d, "p"'] & P \\
P_A \arrow[ur, dotted]
\end{tikzcd}$$

This is equivalent to saying that for any proposition P, the map $P^p : P^{P_A} \to P^A$ induced by precomposing with $p : A \to P_A$ is an iso. When it exists, we shall call such an object a *propositional truncation* of A, and denote it by $A \to [A]$.

Definition 3.6.3. Given a type A, a *propositional truncation* of A is a type $[A]$ equipped with a map $A \to [A]$ such that, for any proposition P, the canonical precomposition map

$$P^{[A]} \to P^A$$

is an isomorphism.

Example 3.6.4. In a category of presheaves $\mathsf{Set}^{\mathbb{C}^{\mathsf{op}}}$ the propositions are exactly the subobjects of 1, by Exercise 3.6.1. But since every map $A \to B$ in presheaves can be factored into an epi followed by a mono $A \twoheadrightarrow M \rightarrowtail B$, every object A has a propositional truncation $A \twoheadrightarrow [A] \rightarrowtail 1$. Moreover, since these factorizations are stable under pullback ($\widehat{\mathbb{C}}$ is regular), the propositional truncation operation $[A]$ commutes with pullback, in the sense that for $B \to 1$ we have $B^*[A] \cong [B^*A]$. More generally, for any $f : Y \to X$ and any $A \to X$, we have

$$f^*[A] \cong [f^*A]\,,$$

as in the following diagram,

$$\begin{tikzcd}
f^*A \arrow[r] \arrow[d, two heads] \arrow[dr, phantom, "\lrcorner", very near start] & A \arrow[d, two heads] \\
{[f^*A]} \arrow[r] \arrow[d, tail] \arrow[dr, phantom, "\lrcorner", very near start] & {[A]} \arrow[d, tail] \\
Y \arrow[r, "f"'] & X
\end{tikzcd}$$

as is seen by applying the foregoing remark to the presheaf category $\widehat{\mathbb{C}}/_X \;\cong\; \widehat{\int_{\mathbb{C}} X}$.

The "stability under pullback" of the operation $A \mapsto [A]$ means that it can be added to dependent type theory as a new type former, because it exists in any context and commutes with substitution. We shall formulate rules for $[A]$ in the next section 3.6.2. The propositional truncation $[A]$ of a type A may be regarded as "erasing (or ignoring) the (computational) content" of A and treating it as a "mere truth-value": either A is inhabited or not, and all inhabitants $a : A$ are identified.

3.6.1 Bracket types

The rules for *bracket types* $[A]$ in dependent type theory are as follows (cf. [?, ?]):

- Formation rule.

$$\frac{A\ \mathtt{type}}{[A]\ \mathtt{type}}$$

- Introduction rules.

$$\frac{a : A}{|a| : [A]} \qquad \frac{a : A,\ b : A}{\mathtt{eq}(a, b) : \mathtt{Eq}_{[A]}(|a|, |b|)}$$

- Elimination rule.

$$\frac{\begin{array}{c} z : [A] \vdash C(z)\ \mathtt{type} \qquad x : A \vdash c(x) : C(|x|) \\ x : A,\ u : C(|x|),\ v : C(|x|) \vdash\ p(x, u, v) : \mathtt{Eq}_{C(|x|)}(u, v) \end{array}}{z : [A] \vdash \mathtt{ind}(z, c, p) : C(z)}$$

- Computation rule.

$$x : A \vdash \mathtt{ind}(|x|, c, p) \equiv c(x) : C(|x|)$$

The Computation rule can be understood semantically as stating that, when it exists, the section $z : [A] \vdash \mathtt{ind}(z, c, p) : C(z)$ makes the following diagram commute.

$$\begin{tikzcd} & C \arrow[d, shift right] \\ A \arrow[ur, "c"] \arrow[r, "|-|"'] & {[A]} \arrow[u, bend right, "\mathtt{ind}"'] \end{tikzcd}$$

And the Elimination rule states that such a section exists if any $u, v : C(|x|)$ are always equal, so that $C(z)$ is a family of propositions.

The simple rules are perhaps easier to understand:

- Simple Elimination rule.

$$\frac{P\ \mathtt{type} \qquad x : A \vdash p(x) : P \qquad u : P,\ v : P \vdash\ q(u, v) : \mathtt{Eq}_P(u, v)}{z : [A] \vdash \mathsf{sind}(z, p, q) : P}$$

- Simple Computation rule.

$$x : A \vdash \mathsf{sind}(|x|, p, q) \equiv p(x) : P$$

The Simple Computation rule simply states that, when it exists, the map $z : [A] \vdash \mathsf{sind}(z, p, q) : P$ makes the following diagram commute.

$$\begin{tikzcd}
A \arrow[r, "p"] \arrow[d, "|-|"'] & P \\
{[A]} \arrow[ur, bend right, "\mathsf{sind}"'] &
\end{tikzcd}$$

And by the Simple Elimination rule, such a map $z : [A] \vdash \mathsf{sind}(z, p, q) : P$ exists whenever P is a proposition. Taken together with the Introduction rule, which says that $[A]$ is always a proposition, this clearly states that the inclusion of the propositions into the types has the propositional truncation operation $[-]$ as a left adjoint,

$$\begin{tikzcd}
\mathsf{Props} \arrow[r, hook, "i"] & \mathsf{Types} \arrow[l, bend left, "{[-]}"]
\end{tikzcd} \qquad [-] \dashv i$$

Exercise 3.6.5. Prove the adjointness between the inclusion of propositions into types and the propositional truncation operation. Why doesn't one need to add an η computation rule to the simple Elimination rule to get the required uniqueness of the eliminator?

3.6.2 Completeness of propositions as types

Bibliography

[AG] C. Angiuli and D. Gratzer. Principles of dependent type theory. Online at `https://carloangiuli.com/courses/b619-sp24/notes.pdf`. Version 2024-11-26.

[AGS17] Steve Awodey, Nicola Gambino, and Kristina Sojakova. Homotopy-initial algebras in type theory. 2017.

[AR94] Jiri Adamek and Jiri Rosicky. *Locally Presentable and Accessible Categories.* Number 189 in London Mathematical Society Lecture Notes. Cambridge University Press, 1994.

[AR11] S. Awodey and F. Rabe. Kripke semantics for Martin-Löf's extensional type theory. *Logical Methods in Computer Science*, 7(3):1–25, 2011.

[Awo00] Steve Awodey. Topological representation of the λ-calculus. *Mathematical Structures in Computer Science*, 10:81–96, 2000.

[Awo10] Steve Awodey. *Category Theory.* Number 52 in Oxford Logic Guides. Oxford University Press, 2010.

[Hof95] Martin Hofmann. Syntax and semantics of dependent types. In *Semantics and logics of computation*, volume 14 of *Publ. Newton Inst.*, pages 79–130. Cambridge University Press, Cambridge, 1995.

[How80] William A. Howard. The formulae-as-types notion of construction. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. 1980. Reprinted from 1969 manuscript.

[HS98] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In *Twenty-five years of constructive type theory (Venice, 1995)*, volume 36 of *Oxford Logic Guides*, pages 83–111. Oxford Univ. Press, New York, 1998.

[Joh03] P.T. Johnstone. *Sketches of an Elephant: A Topos Theory Compendium, 2 vol.s.* Number 43 in Oxford Logic Guides. Oxford University Press, 2003.

[Law63] F. W. Lawvere. Functorial semantics of algebraic theories. Ph.D. thesis, Columbia University, 1963.

[Law70] F.W. Lawvere. Equality in hyperdoctrines and comprehension schema as an adjoint functor. *Proceedings of the AMS Symposium on Pure Mathematics XVII*, pages 1–14, 1970.

[LS88] J. Lambek and P.J. Scott. *Introduction to Higher-Order Categorical Logic*. Cambridge, 1988.

[ML84] Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory.* Bibliopolis, 1984.

[MP00] Ieke Moerdijk and Erik Palmgren. Wellfounded trees in categories. *Annals of Pure and Applied Logic*, 104(1):189–218, 2000.

[Pal03] Erik Palmgren. Groupoids and local cartesian closure. 08 2003. unpublished.

[Sco70] Dana S. Scott. Constructive validity. In M. Laudet, D. Lacombe, L. Nolin, and M. Schützenberger, editors, *Symposium on Automatic Demonstration*, volume 125, pages 237–275. Springer-Verlag, 1970.

[See84] R. A. G. Seely. Locally cartesian closed categories and type theory. *Mathematical Proceedings of the Cambridge Philosophical Society*, 95(1):33, 1984.

[Tai68] William W. Tait. Constructive reasoning. In *Logic, Methodology and Philos. Sci. III (Proc. Third Internat. Congr., Amsterdam, 1967)*, pages 185–199. North-Holland, Amsterdam, 1968.