What is a type theory theory?

Dear David, Mitchell[1], & Owen,

tslil clingman
16th August, 2026

I hope that this letter finds you well. I've decided that i'm no longer content to remain ignorant on the matter of these "type theories" about which you are always talking, so i set out to try to determine just what this business is all about — if you like, what is the theory of type theories?, or perhaps more pointedly, what is a type theory theory? Of course i've read and heard many accounts of the theory of type theories, but all of them seemed to me to be given in "local coordinates" as it were — essentially syntactical in that they dealt principally with contexts, substitutions, and the pullback calculus thereof. I mean no insult in using these terms[2], but as an aesthetic preference none of those sorts of accounts ever seemed satisfactory. What would a "coordinate free" account of type theory look like?

I want to reassure you that i don't mean to write a whole letter whose only conclusion is "a category with properties and structure is a type theory theory in that it exhibits a type theory as its internal language". Rather i want to understand what it means to deal in types and terms as first class, but nevertheless remain agnostic about, if not entirely eschewing of, syntactical commitments like contexts and substitution. I realise that those ideas themselves may be interesting to treat for independent reasons, maybe when i'm older i'll appreciate the desire to study them.

I don't presume that i've done anything original, or even in some philosophical sense correct. If you like, this is a case of outsider art — you represent expertise in logic, (simple) categories, and type theory in contradistinction to my (at best) passing familiarity. Nevertheless i hope that you will find this interesting, failing which your finding it amusing would be just as good.

Warning 1

> We're going to follow established terminological patterns for "X theory" and "X model (of an X theory)", where "X" is some sort of descriptor of the nature of the object under consideration, for example, "essentially algebraic". It's immensely unfortunate that the descriptor for the objects in which we're most interested has the word "theory" occupying its last position.

Warning 2

> Despite being written in a voice which might suggest authority and or certainty, this letter is very much a linearisation of my thoughts. To that end, it's possible that due to my lay understanding of type theory there are grave, perhaps even irreparable errors. While i would certainly like to know of any such, it would be too much for you to spend time on seriously surfacing them. With that in mind, read lightly and briskly!

[1] i just know that there are going to be some prosaic decisions in here that irritate Mitchell, apologies in advance, but it could hardly be avoided

[2] after all, coordinate based calculations are a convenience and have enabled and simplified many types of discourse; arguably type theory is supposed to be such a simplifying syntax

1. A simple approach.

The basic carrier of our intent will be simple categories and simple semi-categories. It's a bit early to say why *semi-categories* are important, but one thing to observe is that the definition of simple category must contain a negation: "for all *non-identity* morphisms . . . " and that negation is at odds with the essentially-algebraic flavour of the framework we're describing.

Def. 1.1 (terrible). A **semi-category** is just like a category but it has no prescribed identity morphisms. Functors of semi-categories are exactly what you expect. Every category is a semi-category in a strict sense: *remove all of the identity morphisms.* We will always mean this when we talk about sub-semi-categories of categories. ⌟

That mapping of categories to semi-categories may not be the one that you were expecting, given the perhaps more natural version of "forget that there were identities". However it is very important that we don't mean that.

With that in mind we can give the definition of simplicity.

Def. 1.2 (Makkai). A semi-category $\mathcal{P}$ is **simple** if it

1. is a finitary direct semi-category (there is a grade function[3] $\operatorname{ob}\mathcal{P} : |\cdot| \longrightarrow \omega$ such that every morphism strictly lowers the grade), and
2. has finite fan-out ($\forall X \in \operatorname{ob}\mathcal{P}$, $|\coprod_{Y\in\operatorname{ob}\mathcal{P}} \mathcal{P}(X,Y)| < \aleph_0$).

A category $\mathcal{C}$ is simple when its strict semi-category is simple. There are no extra conditions on functors between simple (semi-)categories. ⌟

As was likely the original motivation for this definition, simple (semi-)categories are the natural home for gradations of "dependency". The two appearances of finite in the above impose roughly the idea that "every object was the result of a finite construction of finitely many, more primitive objects". Said another way, it will prove very useful to understand simple categories as modelling "types and their dependencies".

It would be humorous to end the letter here, but i'm told that type theories also deal in these things called terms, whence the following definition.

Def. 1.3. A **td-category** comprises the data of a pair $(\mathcal{T}, \mathcal{D})$ where $\mathcal{D}$ is a simple, wide sub-semi-category of the category $\mathcal{T}$. We call the morphisms of $\mathcal{D}$ the **dependency morphisms**, notated $D \nrightarrow D'$ and the morphisms in $\mathcal{T}$ are called the **term morphisms**, notated $X \longrightarrow X'$. Where there is no ambiguity (and frankly even when there is), we will refer to a td-category by a single letter $\mathcal{C} = (\mathcal{T}, \mathcal{D})$.

A **td-functor** $F : (\mathcal{T}, \mathcal{D}) \longrightarrow (\mathcal{T}', \mathcal{D}')$ is what you think it is: a functor $F : \mathcal{T} \longrightarrow \mathcal{T}'$ which factors through the semi-category inclusions $\mathcal{D} \rightarrowtail \mathcal{T}$ and $\mathcal{D}' \rightarrowtail \mathcal{T}'$. We may assemble all of this into the **category** TDCat of td-categories and td-functors. ⌟

Remark 1.4. It's worth pausing to consider a potentially helpful alternate name for dependencies: variables, as used in contexts for TYPE judgements. A map $x : Y \nrightarrow X$ in $\mathcal{D}$ says that we have used a variable $x : X$ in forming the type Y, or in syntax $x : X \vdash Y(x)$ TYPE. This point of view allows us to account for two features.

Firstly, while it is possible to take a variable of any type into a context, there is a seriality to the derivation tree: first we must form the new type (judge it TYPE), and only thereafter may we draw variables from it. We may not draw variables from Y in the derivation of its formation. Thus id_X is not a dependency map, whence $\mathcal{D}$ is a semi-category.

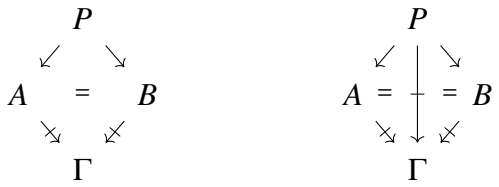
[3] really we mean to use the "canonical" grading function given by, for example, topological sort

Secondly, the inclusion $\mathscr{D} \rightarrowtail \mathscr{C}$ says that every variable of this sort may be viewed as a term. This second feature is a little more difficult to pin down in syntax, but it somehow says that $x\colon X \vdash Y(x)$ TYPE allows us to produce a term of X given a term of Y in a sort of vacuous way, viz., $x\colon X, y : Y(x) \vdash x\colon X$. ◁

With these two basic ingredients, i hope to show how to give a potentially plausible account of what a "type theory theory" and "type theory model" is without ever mentioning limits, contexts, or substitutions. The td-categories will be our "models" of a type theory theory — carrying types, terms and dependencies —, and the finite simple categories will be our "theories", that is, a type theory theory — carrying the various rules.

2. Agricultural elevation.

We will deploy the usual apparatus of category theory in dealing with models, viz., functorial semantics. With an eye to what will come later, consider the following two td-categories. Here and elsewhere we will adopt the convention that unless equations are specified between composites they are not necessarily required.



A functor out of the diagram on the left into another td-category picks a type P and a way of giving terms from that type into types A and B, as well as a joint dependency of the types A and B compatible with P. It is hoped that this is recognisable as part of specifying what it means for a td-category to have products, in that we intend to *quantify* over such situations and explain what must thereafter be possible. Speaking of which, to give a functor out of the diagram on the right is to exhibit P as dependent on Γ in a way that is harmonious with providing the terms of A and B. We wish to suggest that a particular step of defining products may therefore be encoded using the above diagrams as the following sentence:

$$\forall F\colon \text{(left diagram)} \longrightarrow \mathscr{C},\ \left[\exists F'\colon \text{(right diagram)} \longrightarrow \mathscr{C} \text{ st. } F'_{|\text{(left diagram)}} = F\right] .$$

In a way that i have not found succinctly summarisable, we will shortly build a machine parameterised by finite simple categories and operating on td-categories which allows us to build sentences in the language of categories, schematically looking like

$$\forall \ldots, \left[\left[\exists \ldots \text{ st. } \cdots = \cdots \wedge [\text{nest}]\right] \wedge \ldots \wedge \left[\exists \ldots \text{ st. } \cdots = \cdots \wedge [\text{nest}]\right]\right] ,$$

where we have strict alternation between "for all" and "there exists" in depth, and the nesting is finite. This is certainly not the class of *all* finitary and finite sentences in the language of categories—notably missing is disjunction—, but it seems to me that these are the essentially algebraic sentences in some fashion, and suffice for prescribing the "theory" of a given type theory.

What is this machine? Our first component is the notion of a particular kind of lifting problem, which is the apparatus we'll use to create the sentences we saw above. Although what follows could be made more general, for brevity we'll make the narrowest useful definition.

Def. 2.1. Let $\mathcal{S}$ be a category, let $\mathscr{C}$ be a td-category, and fix a functor $T\colon \mathcal{S}^{\mathrm{op}} \longrightarrow \mathrm{TDCat}$. We define **the set of solutions to T-lifting problems through $\mathscr{C}$**, written $\mathrm{Lift}(T, \mathscr{C})$, to be the set of functors $\tilde{T}$ below rendering the diagram commutative.

$$\mathrm{Lift}(T,\mathscr{C}) \coloneqq \left\{ \begin{array}{ccc} & & \mathrm{TDCat}/\mathscr{C} \\ & \overset{\tilde{T}}{\nearrow} & \big\downarrow \pi \\ \mathcal{S}^{\mathrm{op}} & \xrightarrow[T]{} & \mathrm{TDCat} \end{array} \right\}$$

⌟

The idea is that $(\mathcal{S}, T)$ will chronicle the "theory" of the type theory, and quantification and constructions and conditions may be expressed by lifting against π. However, not just any $\mathcal{S}$ will do: "theory" should mean both finite in description, and finitary in construction. Whence $\mathcal{S}$ for us will be a finite simple category.

The only other component required for our machine has been inside simple (semi-)categories the whole time.

Def. 2.2. Let $\mathscr{C}$ be a simple semi-category. A **sieve U** in $\mathscr{C}$ is a sub-semi-category $U \rightarrowtail \mathscr{C}$ with the property that $\forall f \in \mathrm{mor}\,\mathscr{C}$, $[\mathrm{dom}\, f \in \mathrm{ob}\, U \implies f \in \mathrm{mor}\, U]$ — the "down-closed" sub-semi-categories. We collect all sieves into the **lattice of sieves** $\mathrm{Sv}\,\mathscr{C}$.

Given an object $X \in \mathrm{ob}\,\mathscr{C}$, amongst the sieves in $\mathscr{C}$ are

1. the **empty sieve** $\emptyset$,
2. the **principal sieve on** X given by $\downarrow X \coloneqq \{f \in \mathrm{mor}\,\mathscr{C} \mid \mathrm{dom}\, f = X\} \cup \{X\}$[4],
3. the **supporting sieve for** X given by $\partial X \coloneqq \downarrow X \setminus \{X\}$[5], and
4. the entire semi-category $\mathscr{C}$.

There are evident inclusions $\emptyset \leq \partial X \leq \downarrow X \leq \mathscr{C}$. ⌟

Remark 2.3. If $\mathscr{C}$ is a td-category and $X \in \mathrm{ob}\,\mathscr{C}$ then ∂X is something like the "complete context required to judge X TYPE up-to-but-not-including how we use that context for X specifically" whereas $\downarrow X$ contains that final specification. The former has no real syntactic analogue, but the latter is truly $a_1 \colon A_1 \ldots a_n : A_n \vdash X$ TYPE.

The objects ∂X and $\downarrow X$ are naturally semi-categories because we have to explain how dependencies on one type relate to dependencies on other types. Again syntax has no explicit analogue of this, but it's something like the rules of forming telescopes.

In this way, contexts and other operations "in coordinates" are lurking about as derived concepts, but we never treat them explicitly. ◁

Judging by the section title and the context, i hope that you're expecting sheaves to make an appearance, especially as $\mathrm{Sv}\,\mathscr{C}$ has the evident categorical topology on it.

Lem. 2.4. *Let $\mathcal{S}$ be a simple category, let $\mathscr{C}$ be a td-category, and let $T\colon \mathcal{S}^{\mathrm{op}} \longrightarrow \mathrm{TDCat}$ be a functor. The assignment $(U \in \mathrm{ob}\,\mathrm{Sv}\,\mathcal{S}) \mapsto (\mathrm{Lift}(T_{|U^{\mathrm{op}}}, \mathscr{C}) \in \mathrm{ob}\,\mathrm{Set})$ on objects extends to a sheaf $\mathrm{Lift}(T, \mathscr{C})$ on $\mathrm{Sv}\,\mathcal{S}$.*

Proof. Have a think, but basically sieves are downward closed so if a morphism is in the join of sieves it must already be in at least one of the joinands. ■

[4] excuse the notation

[5] here too

To end the section, we can now put a finger on exactly why disjunction is not in our language. Ultimately the data of type theory models comes down to a bunch of td-functors mapping into a fixed td-category. Observe that $|\mathrm{TDCat}(-, \mathbb{1})| = 1$ so that it is impossible to choose a sufficiently clever td-category which would allow us to represent a disjunction.

3. How do i construct thee? Let me quantify the ways.

We have all the pieces we need to begin the construction now. This section is an attempt to elucidate through an abundance of words. The next section tries again, but with pictures.

Def. 3.1. A **type theory theory** comprises the data of a finite simple semi-category $\mathcal{S}$ together with a functor $T\colon \mathcal{S}^{\mathrm{op}} \longrightarrow \mathrm{TDCat}$ whose images are all finitely presented. ⌟

An example of a type theory theory is that of binary products. Here the finite simple semi-category $\mathcal{S}$ will encode various "stages" of the theory of binary products (formation, pairing, β, η, etc) and the dependencies between them: η depends on β and pairing, but pairing depends only on formation. The theory of binary sums might be similarly described, and by taking coproducts we might describe the theory of "binary products and also binary sums" with no axioms relating the two. The finite, simple nature of $\mathcal{S}$ is thus understood as mandating that there are only finitely many stages and moreover each stage may depend on only finitely many, logically prior stages. Viewed through another lens, the finite simple category of a type theory theory is a collection *instructions* and *dependencies*: "first build this", "then build that", "make sure that … agree on …", etc.

Continuing the instruction metaphor, the functor data $T\colon \mathcal{S}^{\mathrm{op}} \longrightarrow \mathrm{TDCat}$ is a specification of implementations: "form a binary product type by mapping out of …", "the requisite hypothesis for the next stage is a functor out of … such that it additionally restricts to …", etc.

We won't dwell on this now, but let us record:

Expectation 1

> There is an appropriate notion of type theory theory morphism involving morphisms of both the finite simple semi-categories as well as, over these, the functors into TDCat.

I had promised sentences using alternately nested quantifiers, but nothing in the above description suggests how this might arise. Indeed this structure is only made manifest through the *use* of type theory theories, by which we mean to say that our notion of type theory model will enforce this interpretation by conjuring quantifiers to match the parity of the grading of constructions. In this way, when looked at schematically, a type theory theory bears a convention for understanding: even grade stages will be quantified over, odd grade stages will give rise to existential demands.

Def. 3.2. A **type theory model** $(M, \mathcal{C})$ **of a type theory theory** $(\mathcal{S}, T)$ comprises the data of a td-category $\mathcal{C}$ as well as a sub-sheaf $M \rightarrowtail \mathrm{Lift}(T, \mathcal{C})$ on $\mathrm{Sv}\,\mathcal{S}$ with the following.

I. The sheaf is inhabited, $M(\emptyset) = \mathrm{Lift}(T, \mathcal{C})(\emptyset)$. [non-trivial]

II. For each $S \in \mathrm{ob}\,\mathcal{S}$ with even grade the following diagram is a pullback [∀]

$$\begin{array}{ccc} M(\downarrow S) & \rightarrowtail & \mathrm{Lift}(T, \mathcal{C})(\downarrow S) \\ {\scriptstyle \text{restr.}}\downarrow & & \downarrow{\scriptstyle \text{restr.}} \\ M(\partial S) & \rightarrowtail & \mathrm{Lift}(T, \mathcal{C})(\partial S) \end{array} \quad .$$

III. For each $S \in \mathrm{ob}\,\mathcal{S}$ with odd grade the following map is split [∃]

$$M(\downarrow S) \xrightarrow{\text{restr.}} M(\partial S) \quad .$$

⌟

It's instructive to step through this definition to understand its implications for a type theory model. Before any other conditions and structure, a type theory model $(M, \mathscr{C})$ is a sub-presheaf of $\mathrm{Lift}(T, \mathscr{C})$ on $\mathrm{Sv}\,\mathcal{S}$ and so specifies a family of lifts of T against $\pi\colon \mathrm{TDCat}^{\mathscr{C}} \to \mathrm{TDCat}$, indexed by down-closed subcategories of $\mathcal{S}$. That is, lifts for steadily later stages $S \in \mathrm{ob}\,\mathcal{S}$ restrict to those specified for earlier stages. This is one of the "axes" imposing structure and constraints here, the other is that being valued in lifts means picking out families of cocones over $\mathscr{C}$.

More than a pre-sheaf, however, M must be a sheaf. The conditions imposed by that roughly state that if a stage $S \in \mathrm{ob}\,\mathcal{S}$ has multiple dependencies $\{S_i\} \subset \mathrm{ob}\,\mathcal{S}$ then the lifts at S are compatible with families of lifts at the S_i such that they agree on their intersections at even earlier stages. Said another way, lifts are coherently built out of smaller data.

The "non-trivial" condition is the prohibition against claiming that $(M, \mathscr{C})$ is a model by supplying *no data*. A quick calculation shows that there are at most two options for $M(\emptyset)$ as $\mathrm{Lift}(T, \mathscr{C}) = \{*\}$, and the empty sheaf satisfies all other conditions, so this is not an exorbitant demand. The more interesting conditions are those predicated on the parity of the grade of the stage $S \in \mathrm{ob}\,\mathcal{S}$.

Let us suppose that $|S| = 0$ so that $\downarrow S = \{S\}$ and $\partial S = \emptyset$, then condition II above combined with I states that $M(\downarrow S) = \mathrm{Lift}(T, \mathscr{C})(\downarrow S)$ — we are required to collect *all* lifts of $T_{|\{S\}}$ against π. At later, even grade stages S the pattern continues: supposing we have chosen lifts "so far" (∂S), continuing to the next stage $(\downarrow S)$ is universal quantification — we must collect *all* lifts of $T_{|\downarrow S}$ which restrict to the data we have thus far accumulated in $M(\partial S)$. In this way objects of even grade act as "$\forall$" quantifiers, the model collects all possible factorisations of $TS_\ell \longrightarrow TS_r$ through $\mathscr{C}$.

Conversely, at an odd stage $S \in \mathrm{ob}\,\mathcal{S}$, condition III above says that continuing to the next stage $(\downarrow S)$ is existential stipulation given the data so far (∂S) — for all coherent collections of factorisations $TS'_\ell \longrightarrow TS'_r$ through $\mathscr{C}$ for $S' \in \mathrm{ob}\,\partial S$, we are required to functionally produce a factorisation $TS_\ell \longrightarrow TS_r$ through $\mathscr{C}$.

In this way the stages $S \in \mathrm{ob}\,\mathcal{S}$ of a type theory theory $(\mathcal{S}, T)$ enforce on type theory models an alternation between quantifying over all "matching conditions" and existentially stipulating a "construction" in $\mathscr{C}$, such that later quantifications vary over earlier constructions and everything is coherent. Both of these "matching conditions" and "constructions" are handled uniformly through the language of factorising td-functors through the carrier of a type theory model.

Expectation 2

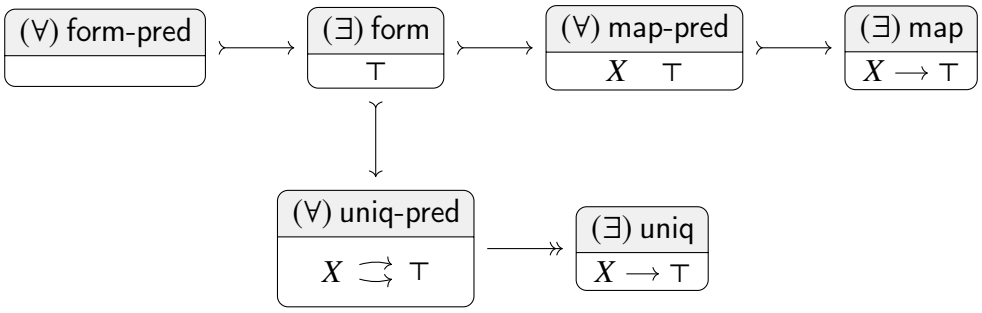
> There is an appropriate notion of type theory model morphism involving morphisms of both the td-category as well as the sheaf. It's possible that these morphisms may have to be of pre-sheaves, or that they will come with conditions.

4. Up and to the right: examples of lifting from T to π.

If prose is not your preferred medium of understanding enablement, or at best a complementary one, good news: it's time for worked examples. In the sub-sections that follow we will be presenting several type theory theories, but to streamline our notation we will elide several details. Chiefly, we will not explicitly state the finite simple semi-category $\mathcal{S}$ or the functor T, but instead give the resulting image in TDCat. Secondarily we will minimally annotate the functors $Fs\colon FS \longrightarrow FS'$ of TDCat trusting you to fill in the details.

4.1. Terminal type

Below is the type theory theory for the terminal type. As this is the first example, we'll walk through this in greater detail than we intend to exhibit in later examples. Fear not the verbiage, though you may be interested in working out what's going on before consulting it.



Recall that a type theory theory is a finite simple semi-category $\mathcal{S}$ together with a functor $T\colon \mathcal{S}^{\mathrm{op}} \longrightarrow \mathrm{TDCat}$. The picture above is our way of communicating both of these pieces of data simultaneously. The finite simple semi-category $\mathcal{S}$ is the opposite of the outer diagram of boxes and arrows (less any of the annotations carried by the arrows), where the objects are named in accordance with the boxes ("form-pred", "form", etc.). The functor T picks out the td-categories given by the insides of the boxes, and the td-functors between these indicated by the morphisms between the boxes (where the annotations are now important). Finally we have notated the parity of the grade of the objects of $\mathcal{S}$ in parentheses with "∀" for even grade, and "∃" for odd grade.

With that, let's walk through what a type theory model for this particular type theory theory must be. First it must be a choice of td-category $\mathcal{C}$ which will be the carrier. Next it must furnish, at first, a non-trivial sub-pre-sheaf M of $\mathrm{Lift}(T, \mathcal{C})$ on $\mathrm{Sv}\,\mathcal{S}$.

With reference to the above picture, sieves on $\mathcal{S}$ are sub-pictures which are "down-closed", following arrows backwards to new boxes must always remain in the sub-picture. Aside from "the whole picture" and "the empty picture", the other two important classes of sieves are the principal ones (for example, the whole top row is the principal sieve on "map") and the supporting sieves on boxes (for example, "form-pred" $\rightarrowtail$ "form" is the supporting sieve on "uniq-pred" as well as on "map-pred").

Unwinding the lifting condition shows that M as a pre-sheaf must assign to each such sieve ("down-closed sub-picture") an inhabited collection of cocones of td-functors into $\mathcal{C}$. The pre-sheaf condition says that earlier cocone collections must include restrictions of later cocone collections to smaller sieves. The more interesting restrictions appear as we work our way deeper into the definition.

That M is required to be a sheaf is a specification of a recipe for "building" M: suppose we have supplied the data of a collection of cocones over $\mathcal{C}$ at $\downarrow$"form", then we may proceed to independently supply the data for $\downarrow$"map-pred" and $\downarrow$"uniq-pred" as long as we enforce that the restrictions of these collections to the point of divergence "form" are the same.

This specification of building M becomes clear when we consider the remaining two conditions on type theory models. At even grades, there is no choice — M must contain all cocones over $\mathscr{C}$ extending those of earlier stages — whereas at odd stages M must functionally determine cocones over $\mathscr{C}$ from the earlier stages.

Applying this to the above diagram, at "form-pred" M is exactly the unique cocone from the empty td-category to $\mathscr{C}$. The stage "form" then mandates that M specify a unique cocone into $\mathscr{C}$ over $\downarrow$"form" ($M\downarrow$"form" $\cong M\downarrow$"form-pred" $= \{*\}$). This amounts to single functor picking out $\top$ in $\mathscr{C}$. Note that $\top$ is not required to be unique, the uniqueness lives at the level of M — for each cocone in ∂"form-pred" (all one of them) there is a unique cocone in ∂"form".

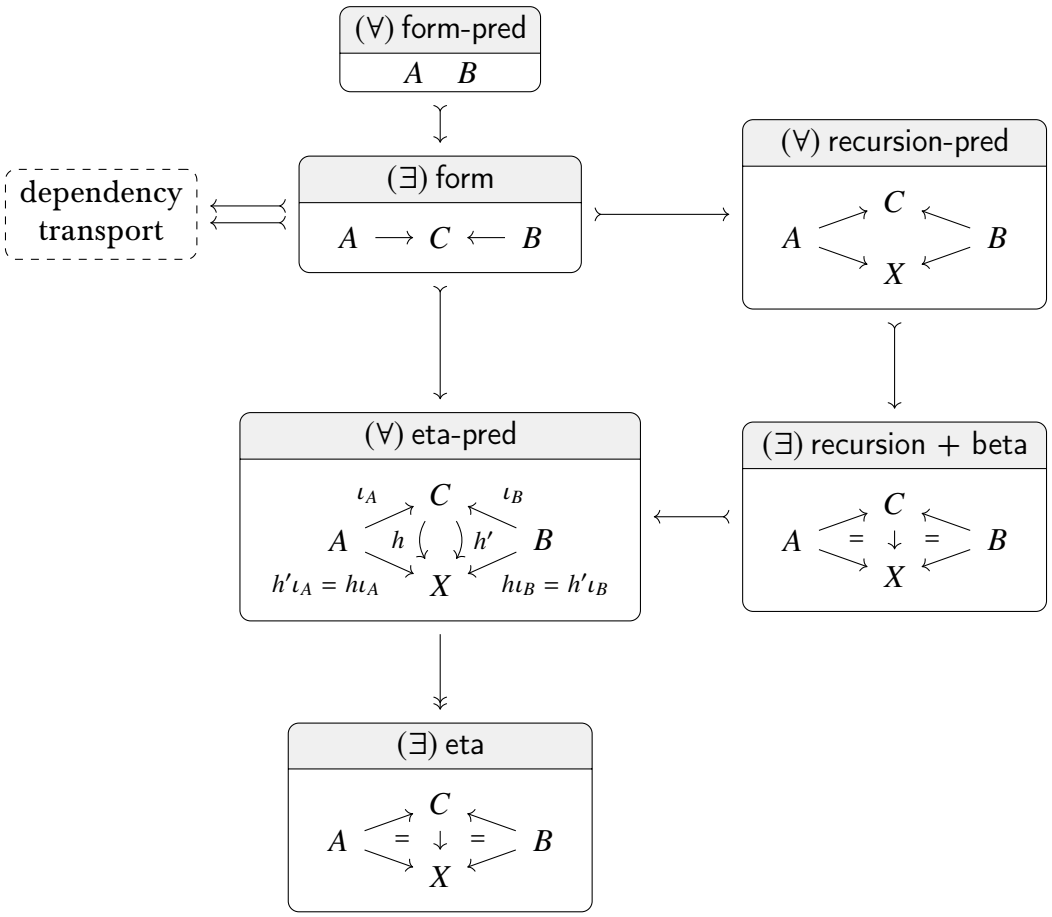
At "uniq-pred" M contains cocones into $\mathscr{C}$ from "form-pred" $\rightarrowtail$ "form" $\rightarrowtail$ "uniq-pred", that is, a functor picking out a pair of objects and parallel morphisms between them. However, because this is an even stage, M is forced to contain all possible choices of these cocones such that they restrict to the previously specified cocones of M at ∂"uniq-pred". That is, M is forced to contain all cocones over $\mathscr{C}$ which pick out parallel morphisms into the $\top$ object that was previously specified.

The last step that we'll look at in detail is "uniq". As this is an odd stage, the data of M is determined completely and precisely by the data at ∂"uniq-pred". That is, for every cocone into $\mathscr{C}$ over that prefix — a choice of parallel arrows into the $\top$ object — there is a uniquely associated cocone extending this to a map in from "uniq". The cocone condition here is crucial, forcing the parallel arrows to be unique.

I'd encourage you to think about the two interacting dimensions here (sieve inclusions and cocones) to get a feeling for the sorts of things that are expressible by this framework.

4.2. Binary coproducts: recursion

The next-least-complicated type theory theory that we're interested in dealing with is that of coproducts with recursion only. Hopefully the below is not very surprising, and we'll attend to "dependency transport" shortly. It's worth pointing out that this is the first time that we'll see a recombination shape in these pictures[6], and we'll discuss that below.



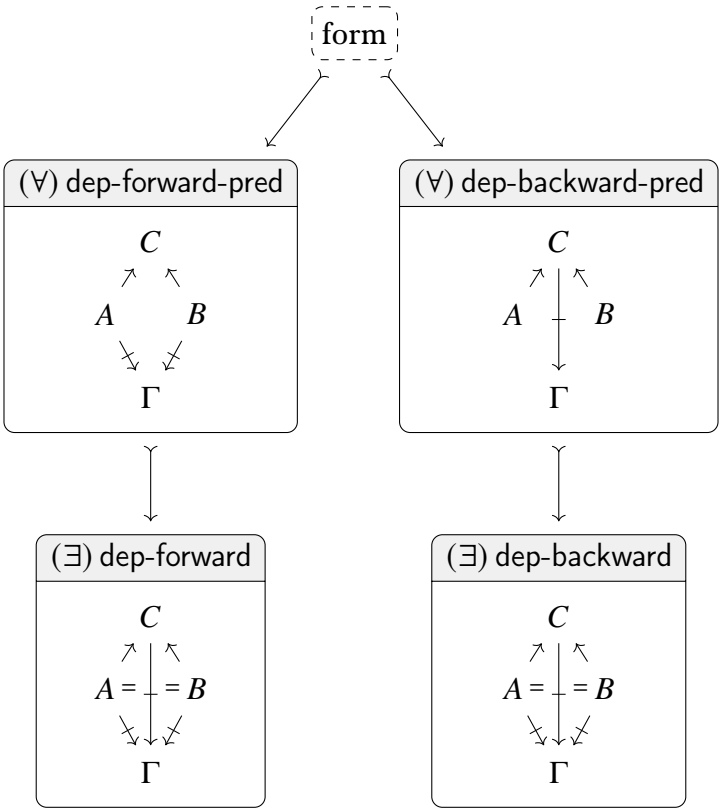
That the even stage "eta-pred" combines two earlier stages imposes our first conjunction (even though it's degenerate):

$$
\begin{aligned}
& M(\downarrow \text{“eta-pred”}) \\
\overset{\text{def.}}{=}\ & M(\partial \text{“eta-pred”}) \times_{\mathrm{Lift}(\partial \text{“eta-pred”})} \mathrm{Lift}(\downarrow \text{“eta-pred”}) \\
\overset{\text{sieve}}{=}\ & M\left(\downarrow \text{“form”} \cup \downarrow \text{“recursion+beta”}\right) \times_{\mathrm{Lift}(\partial \text{“eta-pred”})} \mathrm{Lift}(\downarrow \text{“eta-pred”}) \\
\overset{\text{sheaf}}{=}\ & \left(M(\downarrow \text{“form”}) \times_{M(\downarrow \text{“form”})} M(\downarrow \text{“recursion+beta”})\right) \times_{\mathrm{Lift}(\partial \text{“eta-pred”})} \mathrm{Lift}(\downarrow \text{“eta-pred”})
\end{aligned}
$$

Thus being a type theory model for this type theory theory means that the collection of cocones for $\downarrow$"eta-pred" must be (after simplification) all possible images of "eta-pred" in $\mathscr{C}$ such that the cospan and h (say) come from (respectively) exactly the specifications of the data in "form" *and* "recursion+beta" (which in turn have further agreements between them).

Now for dependency transport. Unlike in the case of the type theory theory of a terminal type, forming the coproduct type of two other types is not necessarily done "in an empty context". Here then is the content of dependency transport.

[6] it's not necessary, but this is a great opportunity to discuss it in a simple and familiar setting



Although we have recursion, given two dependencies as displayed in "dep-forward-pred" nothing about "recursion + beta" allows us to conclude that the exhibited arrow from C must be a dependency arrow if the ones out of A, B were; similarly nothing would otherwise say that pre-composition by an injection takes dependencies to dependencies.

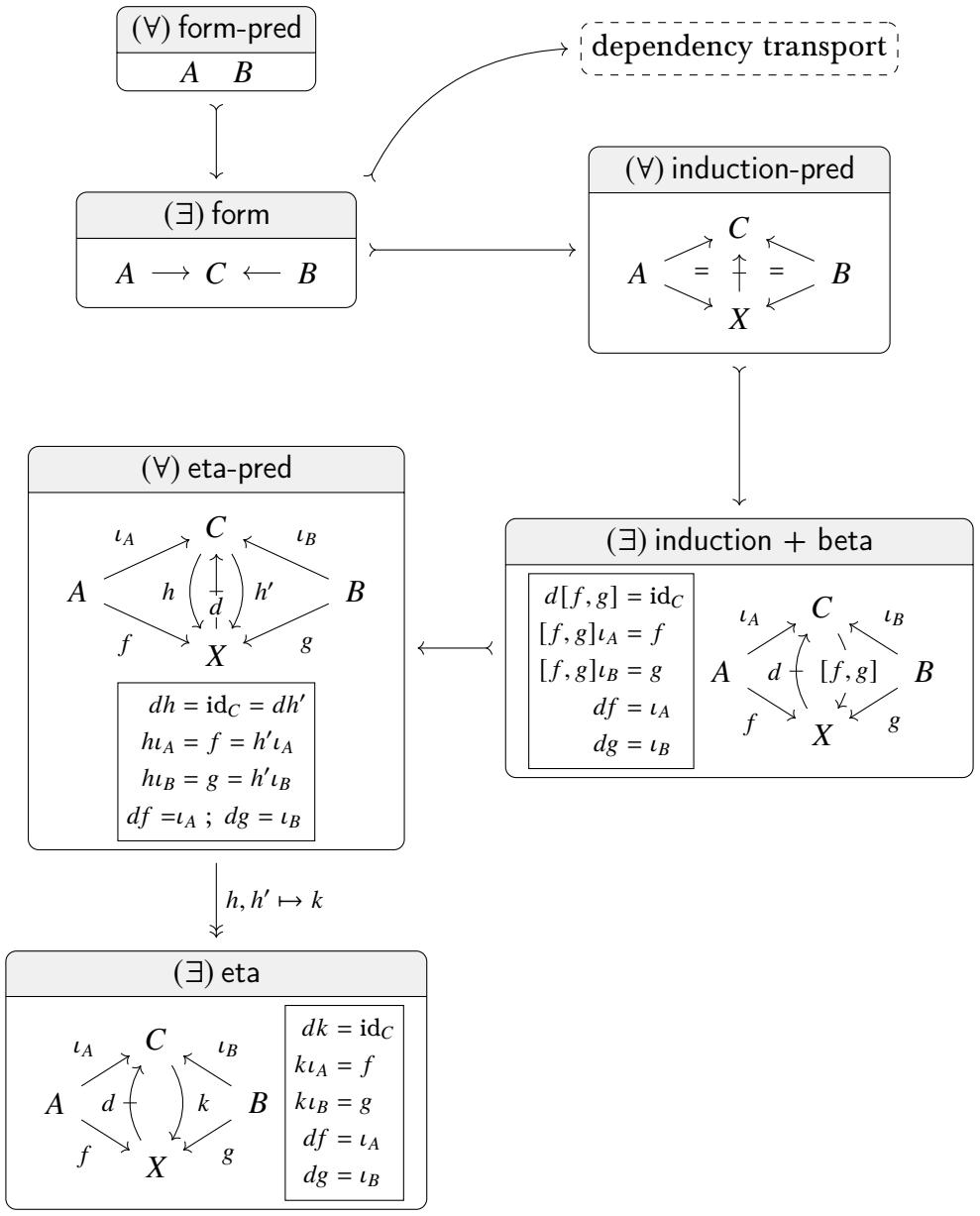
It's worth pointing out that there is a sense in which we have overlapping constructions: because all dependency arrows are in particular arrows, the situation quantified over in "dep-forward-pred" is an example of the situation quantified over by "recursion pred". There is nothing contradictory about this, a type theory model must provide further extensions to "dep-forward" and "recursion + beta". On the other hand, peculiar to coproducts and exploited in this type theory theory, this situation is also not a degree of freedom: because dependency arrows out of C are in particular arrows out of C the rest of the type theory theory applies and uniquely pins these down, so that both paths must agree in any given model. This last fact is also the reason that we haven't tried to articulate any equations between the backward and forward paths in dependency transport — every model validates an isomorphism of semi-categories $(\downarrow A) \times (\downarrow B) \cong \downarrow C$.

Remark 4.1. This dependency structure brings into stark contrast a practical manifestation of the general issue that we were earlier talking about: dependencies must be semi-categories.

Here if identity dependence was admitted, we could select a $\Gamma = C$ in "dep-backward-pred" and observe that all models would have to validate the summands as being dependent upon the sum. Some sort of contradictory behaviour would recur in every example we give for nefarious degenerate dependencies and down that road lies the realisation that we would not be able to quantify over dependencies at all: we would be forced to talk about $\downarrow X \setminus \{\mathrm{id}_X\}$ but that is inaccessible in our framework! ◁

4.3. Binary coproducts: induction

It would be tempting to claim that coproducts with induction generalise coproducts with recursion, but this need not be true in general. Sufficient conditions for this include the presence of weakening, although other derivations may be possible. As such, the categories of type theory models for the two type theory theories (recursion vs. induction) may in general be different.

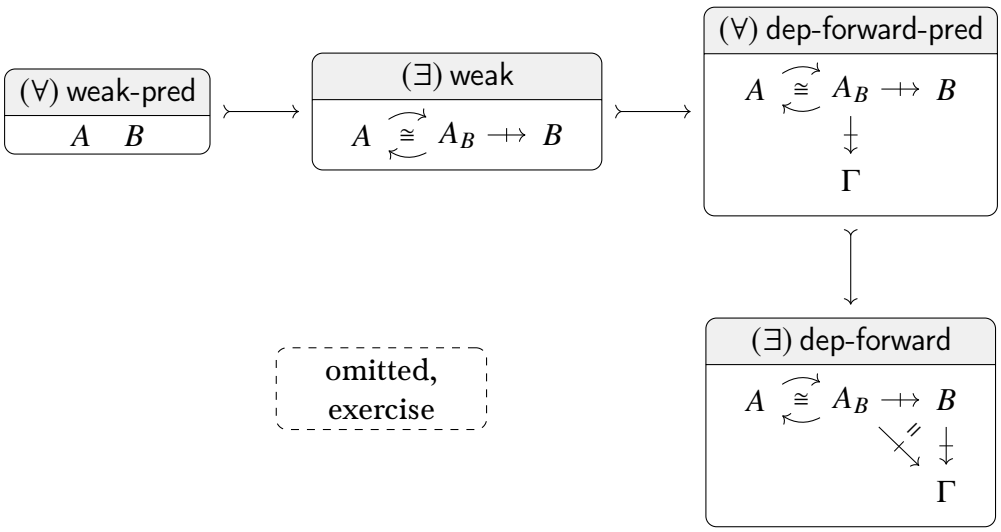


We wish to draw attention to two aspects of the above picture. First, we have omitted the dependency transport component. In this situation we cannot rely on conveniences afforded by recursion and the full picture involves not only the beginning diagrams from the recursion account, but an encoding of the fact that these assignments are mutually inverse and furthermore that they are functorial in dependency composition. I'd encourage you to have a go at producing the pictures for yourself.

Second, we have used what is hopefully a familiar trick to express induction. The target dependent type varying over C has been expressed as $X \twoheadrightarrow C$, and moreover a term in X from C has been expressed as a section of this dependency map. The rest of the type theory theory is essentially the same as the recursion instance, modulo the fact that everything must be predicated on (necessarily term) sections of this dependency map instead of all terms.

4.4. Weakening

It seems topical now to talk about weakening. There are likely a few ways to phrase this, and we have opted here for a sort of "replacement" principle: given two types A, B, we may replace the first by a (necessarily term) isomorphic type A_B which is dependent on B and moreover the dependencies of A_B arise as dependencies of A via the isomorphism as well as the dependencies of B via the dependence $A_B \twoheadrightarrow B$.



Although we produce scant little of this type theory theory, one of the reasons for this excursus is that it is our first confrontation with the lack of disjunction. We'd like to say that "the dependencies of A_B are either of the form … **or** …" but this is inexpressible in our framework! The apparent best that we can do is as we have indicated, "amongst the dependencies of A_B are … **and** …". It's not all bad news, however, for the inexpressibility of this property means that no type theory theory could detect that a type theory model is "improper" in some way by providing A_B with more dependencies than is strictly required.

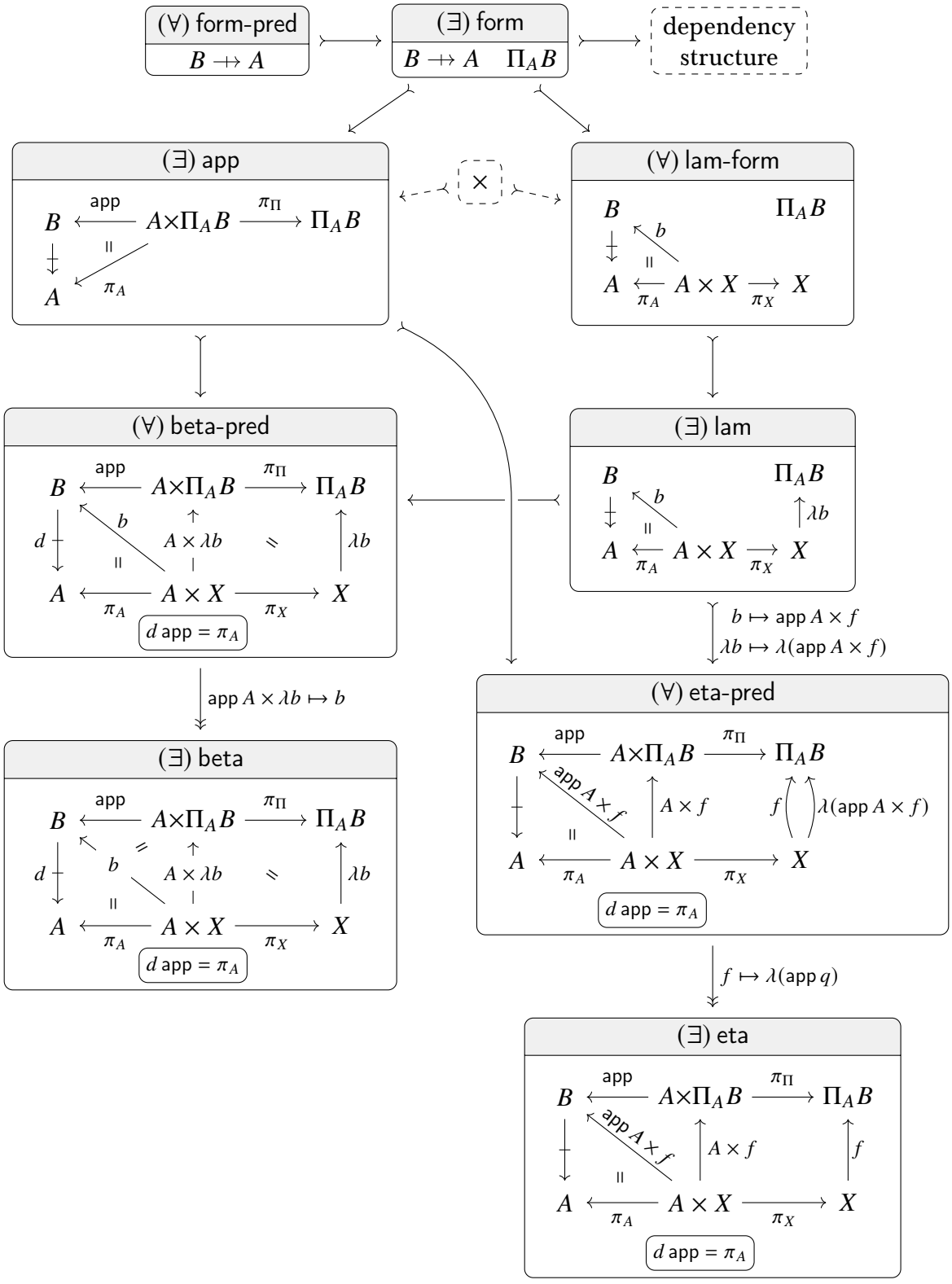
The other reason for this excursus is the following, our first meta-theoretic peek.

Expectation 3

> If $(M_W, \mathscr{C})$ is a type theory model of weakening and $(M_R, \mathscr{C})$ is a type theory model of coproducts with induction, then there is a canonical M_R such that $(M_R, \mathscr{C})$ is a type theory model of coproducts with recursion. The sense of canonical here is meant in some appropriate sense of functorial.

4.5. Dependant products

Dependent products are a marked step up in complexity, because they are the first case where we need to build up a whole "unrelated" type theory theory sitting inside the one we actually want so that we might express things appropriately. In this case, where the picture below indicates "×" we mean that we have developed the type theory theory of binary products with recursion and that that is nested inside our actual type theory theory for dependent products. Thus the various described td-categories below indicate product types and canonical recursion morphisms through their common names, for example, $A \times X$ for a product type of A and X, $\pi_\bullet$ for projection terms, and $A \times f$ for term maps between product types.

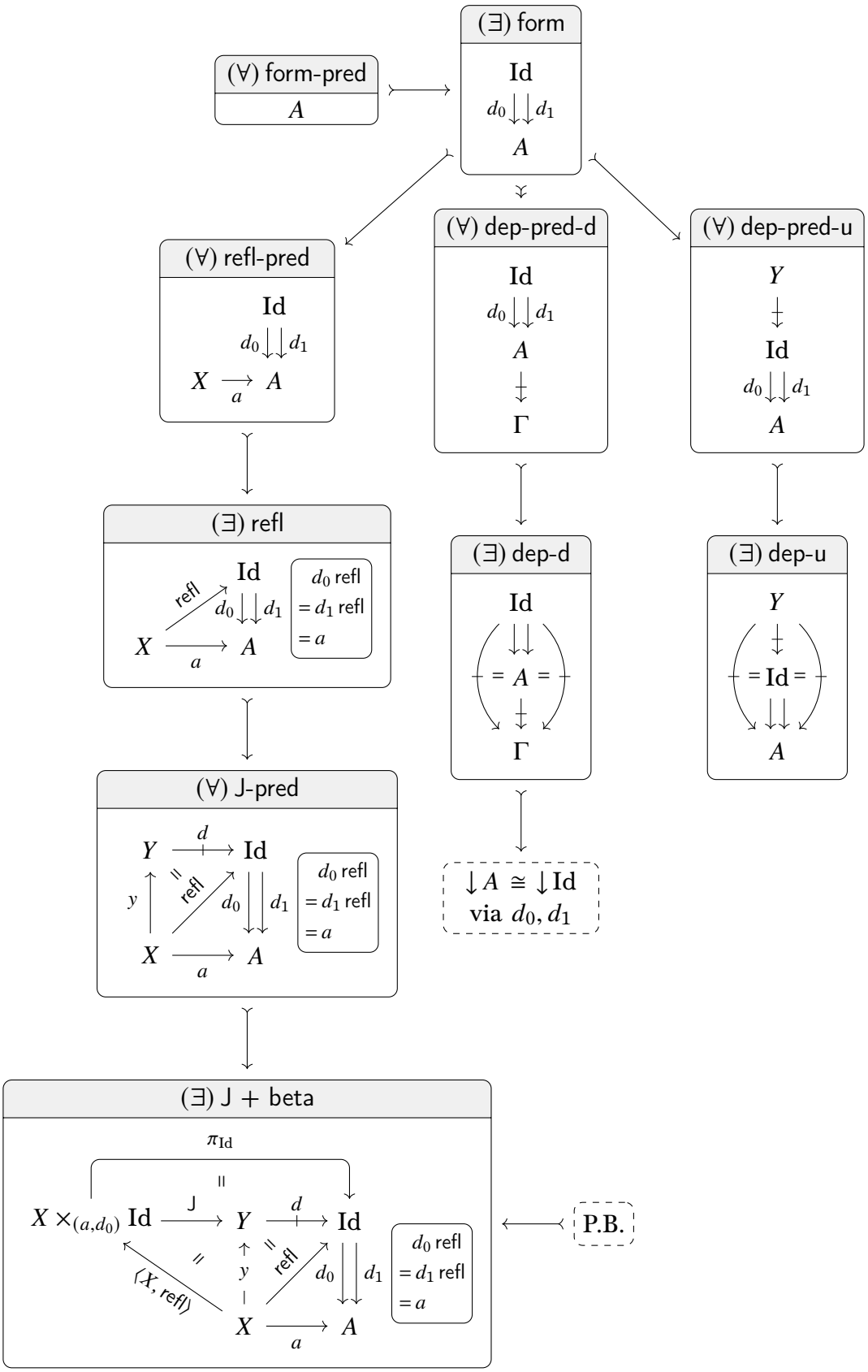


We hope that it will amuse the reader to work out how the sheaf condition and sieve equations give rise to the appropriate conjunctive constraints in the above picture, wherever there occur multiple mappings in. Furthermore on the exercise theme, we have pulled our by-now usual stunt of omitting the dependency structure story. Unlike the cases we have seen so far which ask that dependencies are produced by specified operations, for dependent products there is nothing connecting $\Pi_A B$ to A. As such the story is actually somewhat simpler: we directly quantify over dependencies of the one type and require that (functorially) there are dependencies of the other type, and ask that these inverse functors witness an isomorphism $\downarrow A \cong \downarrow \Pi_A B$.

In the above we have used product types to allow us to express the notion of variable capture/binding: "lam-form" predicates on terms $b(a, x)$ with variable a which are turned into terms $\lambda a.b(a, x)$ without variable a in "lam". We have also leaned on products to express that app is a binary term, and have expressed partial application to a given term f through the usual move of pre-composition by $A \times f$. It seems to me that these are necessary affordances in that no clever expression of dependent products is possible without the digression of developing the type theory theory of binary products, for to speak of terms of multiple arguments or binding all but invokes the very constructions of products in the first place.

4.6. Identity types

Just as in the case of dependent products where we found it somewhere between necessary and convenient to have access to another complete type theory theory, this time we will find it of matching utility to have access to the construction of pullbacks. The presence of this is indicated in our type theory theory below by "P.B.".



A few comments are in order. First we have presented the type theory theory for Paulin-Mohring identity types. The "standard"[7] variant would more-or-less replace the pullback $X\times_{(a,d_0)}$ Id with a wide pullback $X\times_{(a,d_0)}\mathrm{Id}\times_{(d_1,a)}X$ and a few more fidgety details. Once again we have suppressed some details that we feel would be exhausting[8]. In particular, much like in the case of dependent products, pullbacks seem inescapable for a type theory theory of intensional equality for we appear to need to articulate "specialise this type to terms of the form ..." and we have not attempted to present those details here. Finally, apparently new in this example is the "dep-u" development, and this does not appear to follow from any other details about ordinary dependency isomorphism via d_0, d_1.

[7] whose?

[8] for both you and me

5. A typical ending.

I think that 17 pages is stretching the idea of a letter, so let's wrap up with some rapid-fire expectulation and question posing.

Question 1

> What do standard modalities look like as type theory theories? Which are inexpressible in this framework, and what (if anything) is interestingly different about those classes?

Question 2

> In what sense are type theory theories and type theory models essentially algebraic? What can we learn from this connection? Desiderata include abstract nonsense results about limits and filtered colimits of type theory models — with the exception of the terminal type theory model, which usually does not exist (exercise: what is the terminal semi-category and what does this mean for the terminal td-category?).

Expectation 4

> Using the reasonable notion of morphism of type theory theory from an earlier expectation, the category of such TTT has interesting colimits and moreover the functor sending a type theory theory to its category of type theory models $\mathrm{Mod}\colon \mathrm{TTT}^{\mathrm{op}} \longrightarrow \mathrm{Cat}$ is continuous.

Expectation 5

> For a fixed type theory theory $(\mathcal{S}, T)$ the forgetful functor $U\colon \mathrm{Mod}(\mathcal{S}, T) \longrightarrow \mathrm{TDCat}$ which forgets the sheaf M has a left adjoint under some conditions on $(\mathcal{S}, T)$. The left adjoint is constructed through a transfinite operation based on the structure of theory.

Expectation 6

> The type theory theory $\mathcal{M}$ of intensional Martin-Löf satisfies whatever conditions are required for a left adjoint to U to exist, and so in particular admits an initial model.

Question 3

> What is an appropriate notion of syntax for type theory models, and what is the connection to the usual syntax of type theories?

This last question is of particular interest to me because one of the motivating reasons i ended up thinking about type theory theories was to design a pluggable type checker: a single algorithm into which one could insert descriptions of the type theory theories and obtain generic checkers for type theory models.

Now i can finally work on other things,
Thank you,
tslil

This work is licensed under a Creative Commons "Attribution-ShareAlike 4.0 International" license.

