A Category Theoretic View of Contextual Types: from Simple Types to Dependent Types

JASON Z. S. HU, McGill University, Canada
BRIGITTE PIENTKA, McGill University, Canada
ULRICH SCHÖPP, fortiss GmbH, Germany

We describe the categorical semantics for a simply typed variant and a simplified dependently typed variant of Cocon, a contextual modal type theory where the box modality mediates between the weak function space that is used to represent higher-order abstract syntax (HOAS) trees and the strong function space that describes (recursive) computations about them. What makes Cocon different from standard type theories is the presence of first-class contexts and contextual objects to describe syntax trees that are closed with respect to a given context of assumptions. Following M. Hofmann's work, we use a presheaf model to characterise HOAS trees. Surprisingly, this model already provides the necessary structure to also model Cocon. In particular, we can capture the contextual objects of Cocon using a comonad $\flat$ that restricts presheaves to their closed elements. This gives a simple semantic characterisation of the invariants of contextual types (e.g. substitution invariance) and identifies Cocon as a type-theoretic syntax of presheaf models. We further extend this characterisation to dependent types using categories with families and show that we can model a fragment of Cocon without recursor in the Fitch-style dependent modal type theory presented by Birkedal et. al..

CCS Concepts: • **Theory of computation → Modal and temporal logics**; **Type theory**.

Additional Key Words and Phrases: category theory, type theory, contextual types, dependent types

ACM Reference Format:
Jason Z. S. Hu, Brigitte Pientka, and Ulrich Schöpp. 2022. A Category Theoretic View of Contextual Types: from Simple Types to Dependent Types. *ACM Trans. Comput. Logic* 1, 1, Article 1 (January 2022), 37 pages. https://doi.org/10.1145/3545115

1 INTRODUCTION

A fundamental question when defining, implementing, and working with languages and logics is: How do we represent and analyse syntactic structures? Higher-order abstract syntax [Pfenning and Elliott 1988] (or lambda-tree syntax [Miller and Palamidessi 1999]) provides a deceptively simple answer to this question. The basic idea to represent syntactic structures is to map uniformly binding structures in our object language (OL) to the function space in a meta-language thereby inheriting α-renaming and capture-avoiding substitution. In the logical framework LF [Harper et al. 1993], for example, we can define a small functional programming language consisting of functions, function application, and let-expressions using a type `tm` as follows:

```
lam : (tm → tm) → tm.        letv: tm → (tm → tm) → tm.        app : tm → tm → tm.
```

Authors' addresses: Jason Z. S. Hu, zhong.s.hu@mail.mcgill.ca, McGill University, McConnell Engineering Bldg. 3480 University St.,, Montréal, Québec, Canada, H3A 0E9; Brigitte Pientka, bpientka@cs.mcgill.ca, McGill University, McConnell Engineering Bldg. 3480 University St.,, Montréal, Québec, Canada, H3A 0E9; Ulrich Schöpp, schoepp@fortiss.org, fortiss GmbH, Munich, Germany.

1529-3785/2022/1-ART1 $15.00
https://doi.org/10.1145/3545115

The object-language term ($\mathsf{lam}\ x.\ \mathsf{lam}\ y.\ \mathsf{let}\ w = x\ y\ \mathsf{in}\ w\ y$) is then encoded as `lam λx.lam λy. letv (app x y) λw.app w y` using the LF abstractions to model binding. Object-level substitution is modelled through LF application; for instance, the fact that $((\mathsf{lam}\ x.M)\ N)$ reduces to $[N/x]M$ in our object language is expressed as `(app (lam M) N)` reducing to `(M N)`.

This approach is elegant and can offer substantial benefits: we can treat objects equivalent modulo renaming and do not need to define object-level substitution.

However, we not only want to just construct HOAS trees, but also to analyse them and to select sub-trees. This is challenging, as sub-trees are context sensitive. For example, the term `letv (app x y) λw.app w y` only makes sense in a context `x:tm,y:tm`. Moreover, one cannot simply extend LF to allow syntax analysis. If one simply added a recursion combinator to LF, then it could be used to define many functions `M: tm → tm` for which `lam M` would not represent an object-level syntax term [Hofmann 1999].

Contextual types [Gabbay and Nanevski 2013; Nanevski et al. 2008; Pientka 2008] offer a type-theoretic solution to these problems by reifying the typing judgement, i.e. that `letv (app x y) λw. app w y` has type `tm` in the context `x:tm,y:tm`, as a *contextual type* $\lceil x{:}\mathtt{tm}, y{:}\mathtt{tm} \vdash \mathtt{tm}\rceil$. The contextual type $\lceil x{:}\mathtt{tm}, y{:}\mathtt{tm} \vdash \mathtt{tm}\rceil$ describes a set of terms of type `tm` that may contain variables x and y. In particular, the contextual object $\lceil x, y \vdash \mathtt{letv}\ (\mathsf{app}\ x\ y)\ \lambda w.\mathsf{app}\ w\ y\rceil$ has the given contextual type. By abstracting over contexts and treating contexts as first-class, we can now recursively analyse HOAS trees [Pientka 2008; Pientka and Abel 2015; Pientka and Dunfield 2008]. Recently, Pientka et al. [2019] further generalised these ideas and presented a contextual modal type theory, Cocon, where we can mix HOAS trees and computations, i.e. we can use (recursive) computations to analyse and traverse (contextual) HOAS trees and we can embed computations within HOAS trees. This line of work provides a syntactic perspective to the question of how to represent and analyse syntactic structures with binders, as it focuses on decidability of type checking and normalisation. However, its semantics remains not well-understood. What is the semantic meaning of a contextual type? Can we semantically justify the given induction principles? What is the semantics of a first-class context?

While a number of closely related categorical models of abstract syntax with bindings [Fiore et al. 1999; Gabbay and Pitts 1999; Hofmann 1999] were proposed around 2000, the relationship of these models to concrete type-theoretic languages for computing with HOAS structures was tenuous. In this paper, we give a category-theoretic semantics for Cocon. This provides semantic perspective of contextual types and first-class contexts. Maybe surprisingly, the presheaf model introduced by Hofmann [1999] already provides the necessary structure to also model contextual modal type theory. Besides the standard structure of this model, we only rely on two key concepts: a box (necessity) modality which is the same modality discussed by Hofmann [1999, Section 6] and a cartesian closed universe of representables. In the first half of this paper, we focus on the special case of Cocon where the HOAS trees are simply-typed. Concentrating on the simply-typed setting allows us to introduce the main idea without the additional complexity that type dependencies bring with them. The dependently-typed case is reserved in Section 6, in which we study the semantics using a categorical framework for dependent types, categories with families (CwFs). We extend the model in the simply typed case such that the domain category is a CwF, the structure of which is preserved by the Yoneda embedding. In Section 7, we study Cocon's relation with a Fitch-style system and show their similarity.

Our work provides a semantic foundation to Cocon and can serve as a starting point to investigate connections to similar work. First, our work connects Cocon to other work on internal languages for presheaf categories with a $\flat$-modality, such as spatial type theory [Shulman 2018] or crisp type theory [Licata et al. 2018]. Second, it may help to understand the relations of Cocon to type theories that use a modality for metaprogramming and intensional recursion, such as [Kavvos

2017]. While Cocon is built on the same general ideas, a main difference seems to be that Cocon distinguishes between HOAS trees and computations, even though it allows mixed use of them. We hope to clarify the relation by providing a semantical perspective.

This paper is an extended version of the conference paper [Pientka and Schöpp 2020].

2 PRESHEAVES FOR HIGHER-ORDER ABSTRACT SYNTAX

Our work begins with the presheaf models for HOAS of Fiore et al. [1999]; Hofmann [1999]. The key idea of those approaches is to integrate substitution-invariance in the computational universe in a controlled way. For the representation of abstract syntax, one wants to allow only substitution-invariant constructions. For example, `lam M` represents an object-level abstraction if and only if `M` is a function that uses its argument in a substitution-invariant way. For computation with abstract syntax, on the other hand, one wants to allow non-substitution-invariant constructions too. Presheaf categories allow one to choose the desired amount of substitution-invariance.

Let $\mathbb{D}$ be a small category. The presheaf category $\widehat{\mathbb{D}}$ is defined to be the category $\mathrm{Set}^{\mathbb{D}^{\mathrm{op}}}$. Its objects are functors $F\colon \mathbb{D}^{\mathrm{op}} \to \mathrm{Set}$, which are also called *presheaves*. Such a functor F is given by a set $F(\Psi)$ for each object Ψ of $\mathbb{D}$ together with a function $F(\sigma)\colon F(\Phi) \to F(\Psi)$ for any object Φ and $\sigma\colon \Psi \to \Phi$ in $\mathbb{D}$, subject to the functor laws. The intuition is that F defines sets of elements in various $\mathbb{D}$-contexts, together with a $\mathbb{D}$-substitution action. A morphism $f\colon F \to G$ is a natural transformation, which is a family of functions $f_\Psi\colon F(\Psi) \to G(\Psi)$ for any Ψ. This family of functions must be natural, i.e. commute with substitution $f_\Psi \circ F(\sigma) = G(\sigma) \circ f_\Phi$.

For the purposes of modelling higher-order abstract syntax, $\mathbb{D}$ will typically be the term model of some domain-level lambda-calculus. By domain-level, we mean the calculus that serves as the meta-level for object-language encodings. It is the calculus that contains constants like `lam` and `app` from the Introduction. We call it domain-level to avoid possible confusion between different meta-levels later. For simplicity, let us for now use a simply-typed lambda-calculus with functions and products as the domain language. It is sufficient to encode the example from the Introduction and allows us to explain the main idea underlying our approach.

The term model of the simply-typed domain-level lambda-calculus forms a cartesian closed category $\mathbb{D}$. The objects of $\mathbb{D}$ are contexts $x_1{:}A_1, \ldots, x_n{:}A_n$ of simple types. We use Φ and Ψ to range over such contexts. A morphism from $x_1{:}A_1, \ldots, x_n{:}A_n$ to $x_1{:}B_1, \ldots, x_m{:}B_m$ is a tuple $(t_1, \ldots, t_m)$ of terms $x_1{:}A_1, \ldots, x_n{:}A_n \vdash t_i : B_i$ for $i = 1, \ldots, m$. A morphism of type $\Psi \to \Phi$ in $\mathbb{D}$ thus amounts to a (domain-level) substitution that provides a (domain-level) term in context Ψ for each of the variables in Φ. Terms are identified up to $\alpha\beta\eta$-equality. One may achieve this by using a de Bruijn encoding, for example, but the specific encoding is not important for this paper. The terminal object is the empty context, which we denote by $\top$, and the product $\Phi \times \Psi$ is defined by context concatenation. It is not hard to see that any object $x_1{:}A_1, \ldots, x_n{:}A_n$ is isomorphic to an object that is given by a context with a single variable, namely $x_1{:}(A_1 \times \cdots \times A_n)$. This is to say that contexts can be identified with product types. In view of this isomorphism, we shall allow ourselves to consider the objects of $\mathbb{D}$ also as types and vice versa. The category $\mathbb{D}$ is cartesian closed, the exponential of Φ and Ψ being given by the function type $\Phi \to \Psi$ (where the objects are considered as types).

The presheaf category $\widehat{\mathbb{D}}$ is a computational universe that both embeds the term model $\mathbb{D}$ and that can represent computations about it. Note that we cannot just enrich $\mathbb{D}$ with terms for computations if we want to use HOAS. In a simply-typed lambda-calculus with just the constant terms `app: tm → tm → tm` and `lam: (tm → tm) → tm`, each term of type `tm` represents an object-level term. This would not be the true anymore, if we were to allow computations in the domain language, since

one could define `M` to be something like `(λx. if x represents an object-level application then M1 else M2)` for distinct `M1` and `M2`. In this case, `lam M` would not represent an object-level term anymore. If we want to preserve a bijection between the object-level terms and their representations in the domain-language, we cannot allow case-distinction over whether a term represents an object-level an application.

The category $\widehat{\mathbb{D}}$ unites syntax with computations by allowing one to enforce various degrees of substitution-invariance. By choosing objects with different substitution actions, one can control the required amount of substitution-invariance.

In one extreme, a set S can be regarded as a constant presheaf. Define the constant presheaf ΩS by $\Omega S(\Psi) = S$ and $\Omega S(\sigma) = \mathsf{id}$ for all Ψ and σ. Since the substitution action is trivial, a morphism $\Omega S \to \Omega T$ in $\widehat{\mathbb{D}}$ amounts to just a function from set S to set T. Since ΩS is thus essentially just the set S, we shall write just S both for the set S and the presheaf ΩS.

The Yoneda embedding represents the other extreme. For any object Φ of $\mathbb{D}$, the presheaf $\mathrm{y}(\Phi) \colon \mathbb{D}^{\mathrm{op}} \to \mathrm{Set}$ is defined by $\mathrm{y}(\Phi)(\Psi) = \mathbb{D}(\Psi, \Phi)$, which is the set of morphisms from Ψ to Φ in $\mathbb{D}$. The functor action is pre-composition. The presheaf $\mathrm{y}(\Phi)$ should be understood as the type of all domain-level substitutions with codomain Φ. An important example is $y(\mathtt{tm})$. In this case, $\mathrm{y}(\mathtt{tm})(\Psi)$ is the set of all morphisms of type $\Psi \to \mathtt{tm}$ in $\mathbb{D}$. By the definition of $\mathbb{D}$, these correspond to domain-level terms of type `tm` in context Ψ. In this way, the presheaf $\mathrm{y}(\mathtt{tm})$ represents the domain-level terms of type `tm`.

The Yoneda embedding does in fact embed $\mathbb{D}$ into $\widehat{\mathbb{D}}$ fully and faithfully, where the action on morphisms is post-composition. This means that y maps a morphism $\sigma \colon \Psi \to \Phi$ in $\mathbb{D}$ to the natural transformation $\mathrm{y}(\sigma) \colon \mathrm{y}(\Psi) \to \mathrm{y}(\Phi)$ that is defined by post-composing with σ. This definition makes y into a functor $\mathrm{y} \colon \mathbb{D} \to \widehat{\mathbb{D}}$ that is moreover full and faithful: its action on morphisms is a bijection from $\mathbb{D}(\Psi, \Phi)$ to $\widehat{\mathbb{D}}(\mathrm{y}(\Psi), \mathrm{y}(\Phi))$ for any Ψ and Φ. This is because a natural transformation $f \colon \mathrm{y}(\Psi) \to \mathrm{y}(\Phi)$ is, by naturality, uniquely determined by $f_\Psi(\mathsf{id})$, where $\mathsf{id} \in \mathbb{D}(\Psi, \Psi) = \mathrm{y}(\Psi)(\Psi)$, and $f_\Psi(\mathsf{id})$ is an element of $y(\Phi)(\Psi) = \mathbb{D}(\Psi, \Phi)$.

Since $\mathbb{D}$ embeds into $\widehat{\mathbb{D}}$ fully and faithfully, the term model of the domain language is available in $\widehat{\mathbb{D}}$. Consider for example $\mathrm{y}(\mathtt{tm})$. Since y is full and faithful, the morphisms from $\mathrm{y}(\mathtt{tm})$ to $\mathrm{y}(\mathtt{tm})$ in $\widehat{\mathbb{D}}$ are in one-to-one correspondence with the morphisms from `tm` to `tm` in $\mathbb{D}$. These, in turn, are defined to be substitutions and correspond to simply-typed (domain-level) lambda terms with one free variable. This shows that substitution invariance cuts down the morphisms from $\mathrm{y}(\mathtt{tm})$ to $\mathrm{y}(\mathtt{tm})$ in $\widehat{\mathbb{D}}$ just as much as one would like for HOAS encodings.

But $\widehat{\mathbb{D}}$ contains not just a term model of the domain language. It can also represent computations about the domain-level syntax and computations that are not substitution-invariant. For example, arbitrary functions on terms can be represented as morphisms from the constant presheaf $\Omega(\mathrm{y}(\mathtt{tm})(\top))$ to $\mathrm{y}(\mathtt{tm})$. Recall that $\top$ is the empty context, so that $\mathrm{y}(\mathtt{tm})(\top)$ is the set $\mathbb{D}(\top, \mathtt{tm})$, by definition, which is isomorphic to the set of closed domain-level terms of type `tm`. The morphisms from $\Omega(\mathrm{y}(\mathtt{tm})(\top))$ to $\mathrm{y}(\mathtt{tm})$ in $\widehat{\mathbb{D}}$ correspond to arbitrary functions from closed terms to closed terms, without any restriction of substitution invariance.

The restriction to the constant presheaf of closed terms can be generalised to arbitrary presheaves. Define a functor $\flat \colon \widehat{\mathbb{D}} \to \widehat{\mathbb{D}}$ by letting $\flat F$ be the constant presheaf $\Omega(F(\top))$, i.e. $\flat F(\Psi) = F(\top)$ and $\flat F(\sigma) = \mathsf{id}$. Thus, $\flat$ restricts any presheaf to the set of its closed elements. The functor $\flat$ defines a comonad where the counit $\varepsilon_F \colon \flat F \to F$ is the obvious inclusion and the comultiplication $\nu_F \colon \flat F \to \flat\flat F$ is the identity. The latter means that the comonad $\flat$ is idempotent.

The category $\widehat{\mathbb{D}}$ not only embeds $\mathbb{D}$ and allows control over how much substitution invariance is wanted in various constructions, it also is very rich in structure, not least because it also embeds

$\mathcal{S}et$. We will show that $\widehat{\mathbb{D}}$ models contextual types and computations about them. We will gradually introduce the structure of $\widehat{\mathbb{D}}$ that we need to this end. We begin here by noting that $\widehat{\mathbb{D}}$ is cartesian closed, in order to introduce the notation for this structure.

Finite products exist in $\widehat{\mathbb{D}}$ and are constructed pointwise.

$$\top(\Gamma) = \{*\} \qquad\qquad (X \times Y)(\Gamma) = X(\Gamma) \times Y(\Gamma)$$

The Yoneda embedding preserves finite products, i.e. $\mathrm{y}(X \times Y) \cong \mathrm{y}(X) \times \mathrm{y}(Y)$.

The category $\widehat{\mathbb{D}}$ has exponentials. The exponential $(X \Rightarrow Y)$ can be calculated using the Yoneda lemma. We recall that the Yoneda lemma states that $Z(\Gamma)$ is naturally isomorphic to $\widehat{\mathbb{D}}(\mathrm{y}(\Gamma), Z)$. With this, we have:

$$(X \Rightarrow Y)(\Gamma) \;\cong\; \widehat{\mathbb{D}}(\mathrm{y}(\Gamma), X \Rightarrow Y) \;\cong\; \widehat{\mathbb{D}}(\mathrm{y}(\Gamma) \times X, Y)$$

Since y preserves finite products, we have in particular $(\mathrm{y}(A) \Rightarrow \mathrm{y}(B))(\Gamma) \cong \widehat{\mathbb{D}}(\mathrm{y}(\Gamma) \times \mathrm{y}(A), \mathrm{y}(B)) \cong \widehat{\mathbb{D}}(\mathrm{y}(\Gamma \times A), \mathrm{y}(B)) \cong \mathbb{D}(\Gamma \times A, B)$. In the case where $A = B = \mathtt{tm}$, this shows that the exponential $\mathtt{tm} \Rightarrow \mathtt{tm}$ represents terms with an additional bound variable. Given exponentials in $\widehat{\mathbb{D}}$, y also preserves exponentials, i.e. $\mathrm{y}(X \Rightarrow Y) \cong \mathrm{y}(X) \Rightarrow \mathrm{y}(Y)$.

3 INTERNAL LANGUAGE

To explain how $\widehat{\mathbb{D}}$ models higher-order abstract syntax and contextual types, we need to expose more of its structure. Presenting it directly in terms of functors and natural transformations is somewhat laborious and the technical details may obscure the basic idea of our approach. We therefore use dependent types as an internal language for working with $\widehat{\mathbb{D}}$.

It is well-known that presheaf categories like $\widehat{\mathbb{D}}$ furnish a model of a dependent type theory that supports dependent products, dependent sums and extensional identity types, among others, see e.g. [Jacobs 1993]. In this section we outline this dependent type theory and how it is related to $\widehat{\mathbb{D}}$. Since the constructions are largely standard, our aim here is mainly to fix the notation for the later sections.

With the cartesian closed structure defined above, it is already possible to use the simply-typed lambda-calculus for working with $\widehat{\mathbb{D}}$. Morphisms of type $X_1 \times \cdots \times X_n \to Y$ in $\widehat{\mathbb{D}}$ can be considered as terms $x_1 : X_1, \ldots, x_n : X_n \vdash t : Y$ of the simply-typed lambda calculus. The cartesian closed structure of $\widehat{\mathbb{D}}$ is enough to interpret abstraction and application terms. With this correspondence, the simply-typed lambda calculus may be used as a language for defining morphisms $\widehat{\mathbb{D}}$. Notice in particular that morphisms $X_1 \times \cdots \times X_n \to Y_1 \times \cdots \times Y_m$ correspond to a list of terms $(t_1, \ldots, t_m)$ of type $x_1 : X_1, \ldots, x_n : X_n \vdash t_i : Y_i$ for all i. Thus, morphisms $X_1 \times \cdots \times X_n \to Y_1 \times \cdots \times Y_m$ can be considered as substitutions $(t_1/y_1, \ldots, t_m/y_m)$ of terms in context $x_1 : X_1, \ldots, x_n : X_n$ for the variables in $y_1 : Y_1, \ldots, y_m : Y_m$.

The category $\widehat{\mathbb{D}}$ has enough structure to extend this idea to a dependently-typed lambda-calculus in a similar way. Let us first explain how contexts, types and terms relate to the structure of $\widehat{\mathbb{D}}$.

- Contexts: Typing contexts Γ, Δ still correspond to objects of $\widehat{\mathbb{D}}$. Also, the morphisms $\Gamma \to \Delta$ in $\widehat{\mathbb{D}}$ still correspond to substitutions, just like in the simply-typed case.
- Types: Due to type dependencies, the types are not simply objects anymore. For each context Γ, the set of types in context Γ is specified by a set $\widehat{\mathsf{Ty}}(\Gamma)$. The elements of $\widehat{\mathsf{Ty}}(\Gamma)$ do not directly appear in $\widehat{\mathbb{D}}$, but they induce morphisms in $\widehat{\mathbb{D}}$. Each type $X \in \widehat{\mathsf{Ty}}(\Gamma)$ determines an object $\Gamma.X$ and a *projection map* $p_X : \Gamma.X \to \Gamma$. The intention is that $\Gamma.X$ represents the context $\Gamma, x : X$ and that the projection maps represent the weakening substitution $\Gamma, x : X \to \Gamma$.

- Terms: A term $\Gamma \vdash m : X$ appear as morphisms $m : \Gamma \to \Gamma.X$ in $\widehat{\mathbb{D}}$ with the property $\mathrm{id} = p_X \circ m$. In essence it is a substitution for the variables in $\Gamma, x : X$ that is the identity on all variables other than x.

Concretely, we define the set $\widehat{\mathsf{Ty}}(\Gamma)$ to consist of presheaves $\int\Gamma^{\mathrm{op}} \to \mathcal{S}et$, where $\int\Gamma$ is the *category of elements* of Γ, which is defined as follows: the objects of $\int\Gamma$ are the set $\{(\Psi, g) \mid g \in \Gamma(\Psi)\}$ and the morphisms σ between $(\Psi, g) \to (\Phi, g')$ are induced from $\sigma : \Psi \to \Phi$ subject to $g = \Gamma(\sigma, g')$. We can show that this construction does form a category. Intuitively, this construction use morphisms in $\mathbb{D}$ as coherence conditions, which are respected in all other constructions in the presheaf model.

For a presheaf $X : \int\Gamma^{\mathrm{op}} \to \mathcal{S}et$ in $\widehat{\mathsf{Ty}}(\Gamma)$, we define the presheaf $\Gamma.X$ in $\widehat{\mathbb{D}}$ by $(\Gamma.X)(\Phi) = \{(g, x) \mid g \in \Gamma(\Phi),\ x \in X(\Phi, g)\}$ with the canonical action on morphisms. The projection map p_X is just the first projection.

This interpretation of types generalizes the simply-typed case. Any object X of $\widehat{\mathbb{D}}$ can be seen as a simple type. It can be lifted to a presheaf in $\overline{X} : \int\Gamma^{\mathrm{op}} \to \mathcal{S}et$ in $\widehat{\mathsf{Ty}}(\Gamma)$ by letting $\overline{X}(\Phi, g) = X(\Phi)$. Note that with this choice, $\Gamma.\overline{X}$ is identical to $\Gamma \times X$, as one would expect from the simply-typed case.

With type dependencies, one needs a substitution operation on types. For any $\sigma : \Delta \to \Gamma$, there is a type substitution function $(-)\{\sigma\}$ from $\widehat{\mathsf{Ty}}(\Gamma)$ to $\widehat{\mathsf{Ty}}(\Delta)$. It maps $X : \widehat{\mathsf{Ty}}(\Gamma)$ to

$$X\{\sigma\}(\Phi : \mathbb{D}, d : \Delta(\Phi)) := X(\Phi, \sigma(\Phi, d))$$

The definition of $\widehat{\mathsf{Ty}}(\Gamma)$ justifies the intuition of contexts as types of tuples as in the simply-typed case. Consider, for example, a context of the form $(\top.X).Y$ for some types $X \in \widehat{\mathsf{Ty}}(\top)$ and $Y \in \widehat{\mathsf{Ty}}(\top.X)$. The presheaf $(\top.X).Y$ is defined such that the elements of $((\top.X).Y)(\Phi)$ have the form $((*, x), y)$ for some x and y. $*$ is the only element of a uniquely chosen singleton set in $\mathcal{S}et$. One should think of them as representing values of the variables of the context $x : X, y : Y$.

All presheaf categories also support dependent function spaces [Hofmann 1997]. For $X : \widehat{\mathsf{Ty}}(\Gamma)$ and $Y : \widehat{\mathsf{Ty}}(\Gamma.X)$, we write $\widehat{\Pi}(X, Y) : \widehat{\mathsf{Ty}}(\Gamma)$ for the dependent function space. This is defined to be

$$\begin{aligned}\widehat{\Pi}(X, Y)(\Phi : \mathbb{D}, d : \Gamma(\Phi)) := \{f : (\Psi : \mathbb{D})(\delta : \Psi \to \Phi)(x : X(\Psi, \Gamma(\delta, d))) \to Y(\Psi, (\Gamma(\delta, d), x)) \\ \mid \forall \Psi, \delta, x, \Psi', \delta' : \Psi' \to \Psi.\ f(\Psi', \delta \circ \delta', X(\delta', x)) = Y(\delta', f(\Psi, \delta, x))\ \}\end{aligned}$$

$\widehat{\Pi}(X, Y)$ is a set of functions the values of which are coherent under the morphisms of the base category $\mathbb{D}$. We can show that this definition of $\widehat{\Pi}$ types respect type substitutions.

This outlines how the structure of $\widehat{\mathbb{D}}$ relates to dependent type theory, see e.g. [Jacobs 1993] for more details. We will use this type theory as a convenient internal language to work with the structure of $\widehat{\mathbb{D}}$. It is well-known that $\widehat{\mathbb{D}}$ has enough structure to support dependent sums and extensional identity types, among others in its internal dependent type theory, see e.g. [Jacobs 1993]. We do not need to spell out the details of their interpretation.

We use Agda notation for the types and terms of this internal dependent type theory. We write $(x : S) \to T$ for a dependent function type and write $\lambda x : S.m$ and $m\ n$ for the associated lambda-abstractions and applications. As usual, we will sometimes also write $S \to T$ for $(x : S) \to T$ if x does not appear in T. However, to make it easier to distinguish the function spaces at various levels, we will write $(x : S) \to T$ by default even when x does not appear in T. We use $\mathtt{let}\ x = m\ \mathtt{in}\ n$ as an abbreviation for $(\lambda x : T.n)\ m$, as usual. For two terms $m : T$ and $n : T$, we write $m =_T n$ or just $m = n$ for the associated identity type.

In the spirit of Martin-Löf type theory, we will identify the basic types and terms needed for our constructions successively as they are needed. In the following sections, we will expose the structure of $\widehat{\mathbb{D}}$ step by step until we have enough to interpret contextual types.

While much of the structure of $\widehat{\mathbb{D}}$ can be captured by adding rules and constants to standard Martin-Löf type theory, for the comonad $\flat$ such a formulation would not be very satisfactory. The issues are discussed by Shulman [2018, p.7], for example. To obtain a more satisfactory syntax for the comonad, we refine the internal type theory into a modal type theory in which $\flat$ appears as a necessity modality. This approach goes back to Barber and Plotkin [1996]; Benton et al. [1993]; Davies and Pfenning [2001] and is also used by recent work of Licata et al. [2018]; Shulman [2018] and others on working with the $\flat$-modality in type theory.

We summarise here the typing rules for the $\flat$-modality which we will rely on. To control the modality, one uses two kinds of variables. In addition to standard variables $x{:}T$, one has a second kind of so-called *crisp* variables $x{::}T$. Typing judgements have the form $\Delta \mid \Theta \vdash m{:}T$, where Δ collects the crisp variables and Θ collects the ordinary variables. In essence, a crisp variable $x{::}T$ represents an assumption of the form $x{:}\flat T$. The syntactic distinction is useful, since it leads to a type theory that is well-behaved with respect to substitution [Davies and Pfenning 2001; Shulman 2018].

The typing rules are closely related to those in modal type systems [Davies and Pfenning 2001; Nanevski et al. 2008], where Δ is the typing context for modal (global) assumptions and Θ for (local) assumptions, and type systems for linear logic [Barber and Plotkin 1996], where Δ is the typing context for non-linear assumptions and Θ for linear assumptions.

$$\frac{}{\Delta, u{::}T, \Delta' \mid \Theta \vdash u{:}T} \qquad \frac{}{\Delta \mid \Theta, x{:}T, \Theta' \vdash x{:}T} \qquad \frac{\Delta \mid \cdot \vdash m : T}{\Delta \mid \Theta \vdash \mathsf{box}\, m : \flat T}$$

$$\frac{\Delta \mid \Theta \vdash m : \flat T \qquad \Delta, x{::}T \mid \Theta \vdash n : S}{\Delta \mid \Theta \vdash \mathtt{let\ box}\ x = m\ \mathtt{in}\ n : S}$$

Given any term $m{:}\ T$ which only depends on modal variable context Δ, we can form the term $\mathsf{box}\, m{:}\ \flat T$. We have a let-term $\mathtt{let\ box}\ x = m\ \mathtt{in}\ n$ that takes a term $m{:}\ \flat T$ and binds it to a variable $x{::}T$. The rules maintain the invariant that the free variables in a type $\flat T$ are all crisp variables from the crisp context Δ.

The model has other structures. For example, the rules for dependent products are:

$$\frac{\Delta \mid \Theta, x{:}T \vdash m{:}S}{\Delta \mid \Theta \vdash \lambda x{:}T.m : (x{:}T) \to S} \qquad \frac{\Delta \mid \Theta \vdash m{:}(y{:}T) \to S \quad \Delta \mid \Theta \vdash n{:}T}{\Delta \mid \Theta \vdash m\ n{:}[n/y]S}$$

Though the dependent function space is supported by the model, in our interpretation, we only use the simple function space. It is also convenient to have a crisp variant of abstractions and applications [Nanevski et al. 2008]:

$$\frac{\Delta, u{::}T \mid \Theta \vdash m{:}S}{\Delta \mid \Theta \vdash \lambda^{\flat} u{::}T.m : (u{::}T) \to^{\flat} S} \qquad \frac{\Delta \mid \Theta \vdash m{:}(u{::}T) \to^{\flat} S \quad \Delta \mid \cdot \vdash n{:}T}{\Delta \mid \Theta \vdash m\ n{:}[n/u]S}$$

The superscripts $\flat$ of λ and $\to$ indicate that we are referring to the crisp variant. Notice that in the application rule, n is necessarily closed, i.e. the local context of n is empty. These rules allow us to directly operate on crisp variables, and we will interpret computation-level functions of Cocon to this crisp function space. Despite its convenience, the full effect of introducing crisp functions to a comonadic modality type theory is still unclear and we leave its investigation to the future. In this paper, however, we do not make use of its full strength, but just use the syntax as a notation for the semantic interpretation.

When Δ is empty, we shall write just $\Theta \vdash m{:}T$ for $\Delta \mid \Theta \vdash m{:}T$.

Let us outline the categorical interpretation of the modality rules. First recall from above that $\flat$ is a comonad on $\widehat{\mathbb{D}}$. For any type $X \in \widehat{\mathsf{Ty}}(\Delta)$ we define the type $\flat X \in \widehat{\mathsf{Ty}}(\flat\Delta)$ by $(\flat X)(\Phi, d) = X(\top, d)$.

Notice that this is well-defined, since $d \in (\flat\Delta)(\Psi)$ implies $d \in \Delta(\top)$ for all Ψ, by definition of $\flat$. Notice that the definition of $\flat X$ makes $\flat(\Delta.X)$ the same as $\flat\Delta.\flat X$.

Since Δ only contains crisp variables, types in it are represented as $\flat X$. In general, when a type T appears in a context $\Delta \mid \Theta$, it is a type in $\widehat{\mathsf{Ty}}(()\flat\Delta.\Theta)$.

A variable lookup into the crisp context requires application of the counit ϵ. Consider

$$v : \flat\Delta \to \flat\Delta.\flat T$$

which is the variable projection of the crisp context. We further need to unwrap $\flat T$ using $\epsilon : \flat T \to T$. Thus we need another natural transformation:

$$h : \flat\Delta.\flat T \to \flat\Delta.T$$
$$h(\Psi) = (id_{\flat\Delta}, \epsilon_\Psi)$$

Thus we have $h \circ v : \flat\Delta \to \flat\Delta.T$. We can show that this natural transformation is a section of the projection map, due to the first component in h is an identity morphism. General variable lookups in the crisp context can then be obtained by weakening.

To model box, let us first consider a section natural transformation $\flat\Delta \to \flat\Delta.\flat T$ given another section $m : \flat\Delta \to \flat\Delta.T$. Though there might be a more general formulation, we take advantage of the fact that $\flat$ is idempotent in our model, and we can immediately have $\flat m$ to be the desired natural transformation, because $\flat\flat\Delta = \flat\Delta$. The general box with Θ can be obtained by weakening.

The interpretation of $\mathtt{let\ box}\ x = m\ \mathtt{in}\ n$ is relatively easier. We consider as an example the special case where Θ is empty. Given $m : \flat\Delta \to \flat\Delta.\flat T$ and $n : \flat\Delta.\flat T \to \flat\Delta.\flat T.S$, we can just apply m as a substitution to n: $n\{m\} : \flat\Delta \to \flat\Delta.S\{m\}$.

Thus we see that the syntax can be interpreted as categorical constructs. In the rest of the paper, we use syntactic translations to simplify our presentation.

4 FROM PRESHEAVES TO CONTEXTUAL TYPES

Armed with the internal type theory, we can now explore the structure of $\widehat{\mathbb{D}}$.

4.1 A Universe of Representables

For our purposes, the main feature of $\widehat{\mathbb{D}}$ is that it embeds $\mathbb{D}$ fully and faithfully via the Yoneda embedding. In the type theory for $\widehat{\mathbb{D}}$, we may capture this embedding by means of a Tarski-style universe. Such a universe is defined by a type of codes Obj for the objects of $\mathbb{D}$ together with a decoding function that maps these codes into types of the type theory for $\widehat{\mathbb{D}}$.

Let Obj be the set of objects of $\mathbb{D}$. Recall from above, that any set can be considered as a constant presheaf with the trivial substitution action, and thus as a type in the internal type theory of $\widehat{\mathbb{D}}$. The terms of this type Obj represent objects of $\mathbb{D}$. The cartesian closed structure of $\mathbb{D}$ gives us terms unit, times, arrow for the terminal object $\top$, finite products $\times$ and the exponential (function type). We also have a term for the domain-level type tm.

$$\vdash \mathtt{Obj}\ \mathtt{type} \qquad \vdash \mathtt{tm} : \mathtt{Obj} \qquad \vdash \mathtt{times} : (a : \mathtt{Obj}) \to (b : \mathtt{Obj}) \to \mathtt{Obj}$$
$$\vdash \mathtt{unit} : \mathtt{Obj} \qquad \vdash \mathtt{arrow} : (a : \mathtt{Obj}) \to (b : \mathtt{Obj}) \to \mathtt{Obj}$$

Subsequently, we sometimes talk about objects of $\mathbb{D}$ when we intend to describe terms of type Obj (and vice versa).

The morphisms of $\mathbb{D}$ could similarly be encoded as a presheaf with many term constants, but this is in fact not necessary. Instead, we can use the Yoneda embedding to decode elements of Obj into actual types. To this end, we use the following:

$$x : \mathtt{Obj} \vdash \mathtt{El}\ x\ \mathtt{type}$$

The type $\mathtt{El}$ is almost direct syntax for the Yoneda embedding. The interpretation of $\mathtt{El}$ in $\widehat{\mathbb{D}}$, given in detail below, is such that, for any object A of $\mathbb{D}$, the type $\mathtt{El}\,A$ is interpreted by the presheaf $\mathrm{y}(A)$. Such a presheaf is called *representable*. One can think of $\mathtt{El}\,A$ as the type of all morphisms of type $\Psi \to A$ in $\mathbb{D}$ for arbitrary Ψ. Recall from above that a morphism of type $\Psi \to A$ in $\mathbb{D}$ amounts to a domain-level term of type A that may refer to variables in Ψ. In this sense, one should think of $\mathtt{El}\,A$ as a type of domain-level terms of type A, both closed and open ones.

That $\mathtt{El}\,A$ is interpreted by $\mathrm{y}A$ means that all constructions on $\mathtt{El}\,A$ in the internal type theory are guaranteed to be substitution invariant. In particular, since the Yoneda embedding is full and faithful, recall Section 2, the type of functions $(x{:}\,\mathtt{El}\,A) \to \mathtt{El}\,B$ corresponds to the morphisms of type $A \to B$ in $\mathbb{D}$. Any closed term of type $(x : \mathtt{El}\,A) \to \mathtt{El}\,B$ corresponds to such a morphism $A \to B$ in $\mathbb{D}$ and vice versa. This is because $\mathtt{El}\,A$ and $\mathtt{El}\,B$ correspond to $\mathrm{y}A$ and $\mathrm{y}B$ respectively and the naturality requirements in $\widehat{\mathbb{D}}$ enforce substitution-invariance, as outlined in Section 2. The type $(x : \mathtt{El}\,A) \to \mathtt{El}\,B$ thus does not represent arbitrary functions from terms of type A to terms of type B, but only substitution-invariant ones. If a function of this type maps a domain-level variable $x{:}\,A$ (encoded as an element of $\mathtt{El}\,A$) to some term $M{:}\,B$ (encoded as an element of $\mathtt{El}\,B$), then it must map any other $N{:}\,A$ to $[N/x]M$.

In more detail, the interpretation of $\mathtt{Obj} \in \widehat{\mathrm{Ty}}(\top)$ and $\mathtt{El} \in \widehat{\mathrm{Ty}}(\mathtt{Obj})$ of the above types in the internal type theory of $\widehat{\mathbb{D}}$ is given by:

$$\mathtt{Obj}(\Psi, *) = \{\Phi \mid \Phi \text{ is an object of } \mathbb{D}\} \qquad \mathtt{El}\,(\Psi, \Phi) = \mathbb{D}(\Psi, \Phi)$$
$$\mathtt{Obj}(\sigma) : \Phi \mapsto \Phi \qquad \mathtt{El}\,(\sigma) : f \mapsto f \circ \sigma$$

Notice in particular that $(\mathtt{Obj}.\mathtt{El}\,)(\Psi) = \{(\Phi, f) \mid \Phi \text{ is object of } \mathbb{D}, f \in \mathbb{D}(\Psi, \Phi)\}$. If, for any object A in $\mathbb{D}$, we substitute along the corresponding constant function $A{:}\,\top \to \mathtt{Obj}$, then we obtain $(\top.\mathtt{El}\,\{A\})(\Phi) = \{(A, f) \mid f \in \mathbb{D}(\Phi, A)\}$. This presheaf is isomorphic to $\mathrm{y}A$.

We note that, while type dependencies often make it difficult to spell out types directly in terms of the categorical structure of $\widehat{\mathbb{D}}$, type dependencies on constant presheaves like $\mathtt{Obj}$ are relatively easy to work with. This is because $\mathtt{Obj}$ is just a set, so that the naturality constraints of $\widehat{\mathbb{D}}$ are vacuous for functions out of $\mathtt{Obj}$. Instead of working with the dependent type directly, we can just work with all its instances. For example, a term of type $(a{:}\,\mathtt{Obj}) \to (b{:}\,\mathtt{Obj}) \to (x{:}\,\mathtt{El}\,a) \to \mathtt{El}\,b$ is uniquely determined by a family of terms $(x{:}\,\mathtt{El}\,A) \to \mathtt{El}\,B$ indexed by objects A and B in $\mathbb{D}$. We have the following lemma, which states that functions out of $\mathtt{Obj}$ have independent values for all arguments.

Lemma 4.1. *In the internal type theory of $\widehat{\mathbb{D}}$, a closed term $t{:}\,(a{:}\,\mathtt{Obj}) \to X$ is in one-to-one correspondence with a family of closed terms $(t_A)_{A \in \mathtt{Obj}}$ such that $t_A : X[A/a]$. In particular, there is no uniformity condition on this family, i.e. for different objects A and B, the terms t_A and t_B may be arbitrary unrelated terms of types $X[A/a]$ and $X[B/a]$.*

Notice that such a lemma would not be true, e.g., with $\mathtt{El}\,A$ instead of $\mathtt{Obj}$. We have seen above that functions of type $\mathtt{El}\,A \to \mathtt{El}\,B$ correspond to morphisms of type $A \to B$ in $\mathbb{D}$. By definition of $\mathbb{D}$, a morphism $A \to B$ corresponds to a domain-level term $x{:}\,A \vdash t : B$. Such terms are not in one-to-one correspondence with families of closed terms of type B indexed by closed terms of type A.

To summarise this section, by considering the Yoneda embedding as a decoding function $\mathtt{El}$ of a universe á la Tarski, we get access to $\mathbb{D}$ in the internal type theory of $\widehat{\mathbb{D}}$. Since the universe consists of the representable presheaves, we call it the *universe of representables*.

The following lemmas state that the embedding preserves terminal object, binary products and the exponential.

Lemma 4.2. *The internal type theory of* $\widehat{\mathbb{D}}$ *has a term* $\vdash$ `terminal`$:$ `El unit`*, such that* $x =$ `terminal` *holds for any* x`: El unit`*.*

Lemma 4.3. *The internal type theory of* $\widehat{\mathbb{D}}$ *justifies the terms below, such that* `fst (pair` x y`)` $= x$*,* `snd (pair` x y`)` $= y$*,* $z =$ `pair (fst` z`) (snd` z`)` *for all* x, y, z*.*

c`: Obj`, d`: Obj` $\vdash$ `fst: (`z `: El (times` c d`))` $\to$ `El` c

c`: Obj`, d`: Obj` $\vdash$ `snd: (`z `: El (times` c d`))` $\to$ `El` d

c`: Obj`, d`: Obj` $\vdash$ `pair: (`x `: El` c`)` $\to$ `(`y `: El` d`)` $\to$ `El (times` c d`)`

Lemma 4.4. *The internal type theory of* $\widehat{\mathbb{D}}$ *justifies the terms below such that* `arrow-i (arrow-e` f`)` $=$ f *and* `arrow-e (arrow-i` g`)` $= g$ *for all* f, g*.*

c`: Obj`, d`: Obj` $\vdash$ `arrow-e: (`x`: El (arrow` c d`))` $\to$ `(`y`: El` c`)` $\to$ `El` d

c`: Obj`, d`: Obj` $\vdash$ `arrow-i: (`y`: (El` c $\to$ `El` d`))` $\to$ `El (arrow` c d`)`

Proof. Lemmas 4.2 and 4.3 are consequences of the preservation of limits of the Yoneda embedding.

For Lemma 4.4, it suffices, by Lemma 4.1, to establish an isomorphism between `El (arrow` A B`)` and `(`y`: El` A`)` $\to$ `El` B for all objects A and B of $\mathbb{D}$. By definition of `El` , this amounts to preservation of exponentials by y. The goal follows because we have the following isomorphisms, which are natural in Γ:

$$\begin{aligned}(\mathrm{y}A \Rightarrow \mathrm{y}B)(\Gamma) &\cong \widehat{\mathbb{D}}(\mathrm{y}\Gamma, \mathrm{y}A \Rightarrow \mathrm{y}B) \cong \widehat{\mathbb{D}}(\mathrm{y}\Gamma \times \mathrm{y}A, \mathrm{y}B)\\ &\cong \widehat{\mathbb{D}}(\mathrm{y}(\Gamma \times A), \mathrm{y}B) \cong \mathbb{D}(\Gamma \times A, B)\\ &\cong \mathbb{D}(\Gamma, A \Rightarrow B) \cong \widehat{\mathbb{D}}(\mathrm{y}\Gamma, \mathrm{y}(A \Rightarrow B)) \cong \mathrm{y}(A \Rightarrow B)(\Gamma)\end{aligned}$$

□

4.2 Higher-Order Abstract Syntax

The last lemma in the previous section states that `El` $A \to$ `El` B is isomorphic to `El (arrow` A B`)`. This is particularly useful to lift HOAS-encodings from $\mathbb{D}$ to $\widehat{\mathbb{D}}$. For instance, the domain-level term constant `lam: (tm → tm) → tm` gives rise to an element of `El (arrow (arrow tm tm) tm)`. But this type is isomorphic to `(El tm → El tm) → El tm`, by the lemma.

This means that the higher-order abstract syntax constants lift to $\widehat{\mathbb{D}}$:

`app: (`m`: El tm)` $\to$ `(`n`: El tm)` $\to$ `El tm` `lam: (`m`: (El tm → El tm))` $\to$ `El tm`

Once one recognises `El` A as y(A), the adequacy of this higher-order abstract syntax encoding lifts from $\mathbb{D}$ to $\widehat{\mathbb{D}}$ as in Hofmann [1999]. For example, an argument M to `lam` has type `El tm` $\to$ `El tm`, which is isomorphic to `El (arrow tm tm)`. But this type represents (open) domain-level terms t`: tm` $\to$ `tm`. The term `lam` M`: El tm` then represents the domain-level term `lam` t`: tm`, so it just lifts the domain-level.

4.3 Closed Objects

One should think of $\flat T$ as the type of 'closed' elements of T. In particular, $\flat$(`El` A) represents morphisms of type $\top \to A$ in $\mathbb{D}$, recall the definition of $\flat$ from Section 2 and that `El` A corresponds to yA. In the term model $\mathbb{D}$, the morphisms $\top \to A$ correspond to closed domain-language terms of type A. Thus, while `El` A represents both open and closed domain-level terms, $\flat$(`El` A) represents only the closed ones.

This applies also to the type `El` $A \to$ `El` B. We have seen above that `El` $A \to$ `El` B is isomorphic to `El (arrow` A B`)` and may therefore be thought of as containing the terms of type B with a

distinguished variable of type A. But, these terms may contain other free domain language variables. The type $\flat(\mathtt{El}\, A \to \mathtt{El}\, B)$, on the other hand, contains only terms of type B that may contain (at most) one variable of type A.

Restricting to closed objects with the modality is useful because it disables substitution-invariance. For example, the internal type theory for $\widehat{\mathbb{D}}$ justifies a function $\mathtt{is\text{-}lam}\colon (x{:}\flat(\mathtt{El}\ \mathtt{tm})) \to \mathtt{bool}$ that returns $\mathtt{true}$ if and only if the argument represents an object language lambda abstraction. We shall define it in the next section. Such a function cannot be defined with type $\mathtt{El}\ \mathtt{tm} \to \mathtt{bool}$, since it would not be invariant under substitution. Its argument ranges over terms that may be open; which particularly includes domain-level variables. The function would have to return $\mathtt{false}$ for them, since a domain-level variable is not a lambda-abstraction. But after substituting a lambda-abstraction for the variable, it would have to return $\mathtt{true}$, so it could not be substitution-invariant.

We note that the type $\mathtt{Obj}$ consists only of closed elements and that $\mathtt{Obj}$ and $\flat\mathtt{Obj}$ happen to be definitionally equal types (an isomorphism would suffice, but equality is more convenient).

4.4 Contextual Objects

Using function types and the modality, it is now possible to work with contextual objects that represent domain level terms in a certain context, much like in Pientka [2008]; Pientka and Abel [2015]. A contextual type $\lceil \Psi \vdash A \rceil$ is a boxed function type of the form $\flat(\mathtt{El}\ \Psi \to \mathtt{El}\ A)$. It represents domain-level terms of type A with variables from Ψ. Here, we consider the domain-level context Ψ as a term that encodes it. The interpretation will make this precise.

For example, domain-level terms with up to two free variables now appear as terms of type

$$\flat(\mathtt{El}\ ((\mathtt{times}\ (\mathtt{times}\ \mathtt{unit}\ \mathtt{tm})\ \mathtt{tm}) \to \mathtt{El}\ \mathtt{tm}),$$

as the following example illustrates.

$$\begin{array}{ll}\mathtt{box}\ (\mathbin{⊼} u{:}\mathtt{El}\ ((\mathtt{times}\ (\mathtt{times}\ \mathtt{unit}\ \mathtt{tm})\ \mathtt{tm}).\ & \mathtt{let}\ x_1 = \mathtt{snd}\ (\mathtt{fst}\ u)\ \mathtt{in} \\ & \mathtt{let}\ x_2 = \mathtt{snd}\ u\ \mathtt{in} \\ & \quad \mathtt{app}\ (\mathtt{lam}\ (\mathbin{⊼} x{:}\mathtt{El}\ \mathtt{tm}.\ \mathtt{app}\ x_1\ x))\ x_2\ )\end{array}$$

Here, the variables x_1 and x_2 are bound at the meta level, i.e. the internal language. As we will see in the next section, the example interprets the open domain-level term $\mathtt{app}\ (\mathtt{lam}\ (\lambda x.\mathtt{app}\ x_1\ x))\ x_2$ with domain-level variables $x_1{:}\mathtt{tm}$ and $x_2{:}\mathtt{tm}$.

This representation integrates substitution as usual. For example, given crisp variables $m{::}\mathtt{El}\ (\mathtt{times}\ c\ \mathtt{tm}) \to \mathtt{tm}$ and $n{::}\mathtt{El}\ c \to \mathtt{tm}$ for contextual terms, the term $\mathtt{box}\ (\mathbin{⊼} u{:}\mathtt{El}\ c.\ m\ (\mathtt{pair}\ u\ (n\ u)))$ represents substitution of n for the last variable in the context of m.

For working with contextual objects, it is convenient to lift the constants $\mathtt{app}$ and $\mathtt{lam}$ to contextual types.

$$c{:}\mathtt{Obj} \vdash \mathtt{app}'\colon \flat(\mathtt{El}\ c \to \mathtt{El}\ \mathtt{tm}) \to \flat(\mathtt{El}\ c \to \mathtt{El}\ \mathtt{tm}) \to \flat(\mathtt{El}\ c \to \mathtt{tm})$$
$$c{:}\mathtt{Obj} \vdash \mathtt{lam}'\colon \flat(\mathtt{El}\ (\mathtt{times}\ c\ \mathtt{tm}) \to \mathtt{El}\ \mathtt{tm}) \to \flat(\mathtt{El}\ c \to \mathtt{El}\ \mathtt{tm})$$

These terms are defined by:

$$\begin{array}{lcl}\mathtt{app}' & := & \mathbin{⊼} m, n.\ \mathtt{let\ box}\ m' = m\ \mathtt{in\ let\ box}\ n' = n\ \mathtt{in} \\ & & \qquad \mathtt{box}\ (\mathbin{⊼} u{:}\mathtt{El}\ c.\ \mathtt{app}\ (m'\ u)\ (n'\ u)) \\ \mathtt{lam}' & := & \mathbin{⊼} m.\ \mathtt{let\ box}\ m' = m\ \mathtt{in\ box}\ (\mathbin{⊼} u{:}\mathtt{El}\ c.\ \mathtt{lam}\ (\mathbin{⊼} x{:}\mathtt{El}\ \mathtt{tm}.\ \ m'\ (\mathtt{pair}\ u\ x)))\end{array}$$

A contextual type for domain-level variables (as opposed to arbitrary terms) can be defined by restricting the function space in $\flat(\mathtt{El}\ \Psi \to \mathtt{El}\ A)$ to consist only of projections. Projections are functions of the form $\mathtt{snd} \circ \mathtt{fst}^k$, where we write $\mathtt{fst}^k$ for the k-fold iteration $\mathtt{fst} \circ \cdots \circ \mathtt{fst}$. Let us

write $\mathtt{El}\,\Psi \to_v \mathtt{El}\,A$ for the subtype of $\mathtt{El}\,\Psi \to \mathtt{El}\,A$ consisting only of projections. The contextual type $\flat(\mathtt{El}\,\Psi \to_v \mathtt{El}\,A)$ is then a subtype of $\flat(\mathtt{El}\,\Psi \to \mathtt{El}\,A)$.

With these definitions, we can express a primitive recursion scheme for contextual types. We write it in its general form where the result type A can possibly depend on x. This is only relevant for the dependently typed case; in the simply typed case, the only dependency is on c.

Lemma 4.5. *Let* $c{:}\mathtt{Obj},\ x{:}\flat(\mathtt{El}\,c \to \mathtt{El}\,\mathtt{tm}) \vdash A\,c\,x$ *type and define:*

$X_{\mathtt{var}} := (c{:}\mathtt{Obj}) \to (x{:}\flat(\mathtt{El}\,c \to_v \mathtt{El}\,\mathtt{tm})) \to A\,c\,x$

$X_{\mathtt{app}} := (c{:}\mathtt{Obj}) \to (x,y{:}\flat(\mathtt{El}\,c \to \mathtt{El}\,\mathtt{tm})) \to A\,c\,x \to A\,c\,y \to A\,c\,(\mathtt{app}'\ x\ y)$

$X_{\mathtt{lam}} := (c{:}\mathtt{Obj}) \to (x{:}\flat(\mathtt{El}\,(\mathtt{times}\,c\,\mathtt{tm}) \to \mathtt{El}\,\mathtt{tm})) \to A\,(\mathtt{times}\,c\,\mathtt{tm})\,x \to A\,c\,(\mathtt{lam}'\,x)$

Then, $\widehat{\mathbb{D}}$ *justifies a term*

$\vdash \mathtt{rec}{:}\ X_{\mathtt{var}} \to X_{\mathtt{app}} \to X_{\mathtt{lam}} \to (c{:}\mathtt{Obj}) \to (x{:}\flat(\mathtt{El}\,c \to \mathtt{El}\,\mathtt{tm})) \to A\,c\,x$

such that the following equations are valid.

$$\begin{array}{lcll} \mathtt{rec}\ t_{\mathtt{var}}\ t_{\mathtt{app}}\ t_{\mathtt{lam}}\ c\ x & = & t_{\mathtt{var}}\ c\ x & \text{if } x{:}\flat(\mathtt{El}\,c \to_v \mathtt{El}\,\mathtt{tm}) \\ \mathtt{rec}\ t_{\mathtt{var}}\ t_{\mathtt{app}}\ t_{\mathtt{lam}}\ c\ (\mathtt{app}'\ s\ t) & = & t_{\mathtt{app}}\ c\ s\ t & \\ \mathtt{rec}\ t_{\mathtt{var}}\ t_{\mathtt{app}}\ t_{\mathtt{lam}}\ c\ (\mathtt{lam}'\ s) & = & t_{\mathtt{lam}}\ c\ s & \end{array}$$

outline. To outline the proof idea, note first that a function of type $(c{:}\mathtt{Obj}) \to (x{:}\flat(\mathtt{El}\,c \to \mathtt{El}\,\mathtt{tm})) \to A\,c\,x$ in $\widehat{\mathbb{D}}$, corresponds to an inhabitant of $A\,\Phi\,t$ for each concrete object Φ of $\mathbb{D}$ and each inhabitant $t : \flat(\mathtt{El}\,\Phi \to \mathtt{El}\,\mathtt{tm})$. This is because naturality constraints for boxed types are vacuous (and $\mathtt{Obj} = \flat\mathtt{Obj}$). Next, note that inhabitants of $\flat(\mathtt{El}\,\Phi \to \mathtt{El}\,\mathtt{tm})$ correspond to domain-level terms of type $\mathtt{tm}$ in context Φ up to $\alpha\beta\eta$-equality. We can perform a case-distinction on whether it is a variable, abstraction or application and depending on the result use $t_{\mathtt{var}}$, $t_{\mathtt{app}}$ or $t_{\mathtt{lam}}$ to define the required inhabitant of $A\,\Phi\,t$. □

As a simple example for `rec`, we can define the function `is-lam` discussed above by $\mathtt{rec}\ (\lambda c, x.\,\mathtt{false})\ (\lambda c, x, y, r_x, r_y.\,\mathtt{false})\ (\lambda c, x, r_x.\,\mathtt{true})$.

5 SIMPLE CONTEXTUAL MODAL TYPE THEORY

We have outlined informally how the internal dependent type theory of $\widehat{\mathbb{D}}$ can model contextual types. In this section, we make this precise by giving the interpretation of Cocon [Pientka et al. 2019], a contextual modal type theory where we can work with contextual HOAS trees and computations about them, into $\widehat{\mathbb{D}}$. We will focus here on a simply-typed version of Cocon where we use a simply-typed domain-language with constants `app` and `lam` and also only allow computations about HOAS trees, but do not consider, for example, universes. Concentrating on a stripped down, simply-typed version of Cocon allows us to focus on the essential aspects, namely how to interpret domain-level contexts and domain-level contextual objects and types semantically. The generalisation to a dependently typed domain-level such as LF in Section 6 will be conceptually straightforward, although more technical. Handling universes is an orthogonal issue.

We first define our simply-typed domain-level with the type `tm` and the term constants lam and app (see Fig. 1). Following Cocon, we allow computations to be embedded into domain-level terms via unboxing. The intuition is that if a program t promises to compute a value of type $\lceil x{:}\mathtt{tm}, y{:}\mathtt{tm} \vdash \mathtt{tm}\rceil$, then we can embed t directly into a domain-level object writing $\mathtt{lam}\ \lambda x.\mathtt{lam}\ \lambda y.\mathtt{app}\ \lfloor t \rfloor\ x$, unboxing t. Domain-level objects (resp. types) can be packaged together with their domain-level context to form a contextual object (resp. type). Domain-level contexts are formed as usual, but may contain context variables to describe a yet unknown prefix. Last, we include domain-level substitutions that allow us to move between domain-level contexts. The compound substitution σ, M extends the substitution σ with domain $\widehat{\Psi}$ to a substitution with domain $\widehat{\Psi}, x$, where M replaces

Domain-level types	A, B	$::= \mathtt{tm} \mid A \to B$
Domain-level terms	M, N	$::= \lambda x.M \mid M\,N \mid x \mid \mathsf{lam} \mid \mathsf{app} \mid \lfloor t \rfloor_\sigma$
Domain-level contexts	Ψ, Φ	$::= \cdot \mid \psi \mid \Psi, x{:}A$
Domain-level context (erased)	$\widehat{\Psi}, \widehat{\Phi}$	$::= \cdot \mid \psi \mid \widehat{\Psi}, x$
Domain-level substitutions	σ	$::= \cdot \mid \mathsf{wk}_{\widehat{\Psi}} \mid \sigma, M$
Contextual types	T	$::= \Psi \vdash A \mid \Psi \vdash_v A$
Contextual objects	C	$::= \widehat{\Psi} \vdash M$
Domain of discourse	$\check{\tau}$	$::= \tau \mid \mathtt{ctx}$
Types and Terms	$\tau, \mathcal{I}$	$::= \lceil T \rceil \mid (y : \check{\tau}_1) \Rightarrow \tau_2$
	t, s	$::= y \mid \lceil C \rceil \mid \mathsf{rec}^{\mathcal{I}}\ \vec{\mathcal{B}}\ \Psi\ t \mid \mathsf{fn}\ y \Rightarrow t \mid t_1\ t_2$
Branches	$\mathcal{B}$	$::= \Gamma \mapsto t$
Contexts	Γ	$::= \cdot \mid \Gamma, y : \check{\tau}$

Fig. 1. Syntax of Cocon with a fixed simply-typed domain tm

x. Following Nanevski et al. [2008]; Pientka et al. [2019], we do not store the domain (like $\widehat{\Psi}$) in the substitution, it can always be recovered before applying the substitution. We also include *weakening substitution*, written as $\mathsf{wk}_{\widehat{\Psi}}$, to describe the weakening of the domain Ψ to $\Psi, \overrightarrow{x{:}A}$. Weakening substitutions are necessary, as they allow us to express the weakening of a context variable ψ. Identity is a special form of the $\mathsf{wk}_{\widehat{\Psi}}$ substitution, which follows immediately from the typing rule of $\mathsf{wk}_{\widehat{\Psi}}$. Composition is admissible.

We summarise the typing rules for domain-level terms and types in Fig. 2. We also include typing rules for domain-level contexts. Note that since we restrict ourselves to a simply-typed domain-level, we simply check that A is a well-formed type. We remark that equality for domain-level terms and substitution is modulo $\beta\eta$. In particular, $\lfloor \lceil \widehat{\Phi} \vdash N \rceil \rfloor_\sigma$ reduces to $[\sigma]N$.

$\boxed{\Gamma; \Psi \vdash M : A}$ Term M has type A in domain-level context Ψ and context Γ

$$\frac{\Gamma \vdash \Psi : \mathtt{ctx} \quad x{:}A \in \Psi}{\Gamma; \Psi \vdash x : A} \qquad \frac{\Gamma \vdash \Psi : \mathtt{ctx}}{\Gamma; \Psi \vdash \mathsf{lam} : (\mathtt{tm} \to \mathtt{tm}) \to \mathtt{tm}} \qquad \frac{\Gamma \vdash \Psi : \mathtt{ctx}}{\Gamma; \Psi \vdash \mathsf{app} : \mathtt{tm} \to \mathtt{tm} \to \mathtt{tm}}$$

$$\frac{\Gamma; \Psi \vdash M : A \to B \quad \Gamma; \Psi \vdash N : A}{\Gamma; \Psi \vdash M\,N : B} \qquad \frac{\Gamma; \Psi, x{:}A \vdash M : B}{\Gamma; \Psi \vdash \lambda x.M : A \to B} \qquad \frac{\Gamma \vdash t : \lceil \Phi \vdash A \rceil \text{ or } \Gamma \vdash t : \lceil \Phi \vdash_v A \rceil \quad \Gamma; \Psi \vdash \sigma : \Phi}{\Gamma; \Psi \vdash \lfloor t \rfloor_\sigma : A}$$

$\boxed{\Gamma; \Phi \vdash \sigma : \Psi}$ Substitution σ provides a mapping from the (domain) context Ψ to Φ

$$\frac{\Gamma \vdash \Psi, \overrightarrow{x{:}A} : \mathtt{ctx}}{\Gamma; \Psi, \overrightarrow{x{:}A} \vdash \mathsf{wk}_{\widehat{\Psi}} : \Psi} \qquad \frac{\Gamma \vdash \Phi : \mathtt{ctx}}{\Gamma; \Phi \vdash \cdot : \cdot} \qquad \frac{\Gamma; \Phi \vdash \sigma : \Psi \quad \Gamma; \Phi \vdash M : A}{\Gamma; \Phi \vdash \sigma, M : \Psi, x{:}A}$$

$\boxed{\Gamma \vdash \Psi : \mathtt{ctx}}$ Domain-level context Ψ is well-formed

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

Fig. 2. Typing Rules for Domain-level Terms, Substitutions, Contexts

In our grammar, we distinguish between the contextual type $\Psi \vdash A$ and the more restricted contextual type $\Phi \vdash_v A$ which characterises only variables of type A from the domain-level context

Φ. We give here two sample typing rules for $\Phi \vdash_v A$ which are the ones used most in practice to illustrate the main idea. We embed contextual objects into computations via the modality. Computation-level types include boxed contextual types, $\lceil \Phi \vdash A \rceil$, and function types, written as $(y : \check{\tau}_1) \Rightarrow \tau_2$. We overload the function space and allow as domain of discourse both computation-level types and the schema ctx of domain-level contexts, although only in the latter case y can occur in τ_2. We use $\mathsf{fn}\ y \Rightarrow t$ to introduce functions of both kinds. We also overload function application $t\ s$ to eliminate function types $(y : \tau_1) \Rightarrow \tau_2$ and $(y : \mathsf{ctx}) \Rightarrow \tau_2$, although in the latter case s stands for a domain-level context. We separate domain-level contexts from contextual objects, as we do not allow functions that return a domain-level context.

The recursor is written as $\mathsf{rec}^{\mathcal{I}}\ \vec{\mathcal{B}}\ \Psi\ t$. Here, t describes a term of type $\lceil \Psi \vdash \mathtt{tm} \rceil$ that we recurse over and $\vec{\mathcal{B}}$ describes the different branches that we can take depending on the value computed by t. As is common when we have dependencies, we annotate the recursor with the typing invariant $\mathcal{I}$. Here, we consider only the recursor over domain-level terms of type $\mathtt{tm}$. Hence, we annotate it with $\mathcal{I} = (\psi : \mathsf{ctx}) \Rightarrow (y : \lceil \psi \vdash \mathtt{tm} \rceil) \Rightarrow \tau$. To check that the recursor $\mathsf{rec}^{\mathcal{I}}\ \mathcal{B}\ \Psi\ t$ has type $[\Psi/\psi]\tau$, we check that each of the three branches has the specified type $\mathcal{I}$. In the base case, we may assume in addition to $\psi : \mathsf{ctx}$ that we have a variable $p : \lceil \psi \vdash_v \mathtt{tm} \rceil$ and check that the body has the appropriate type. If we encounter a contextual object built with the domain-level constant app, then we choose the branch b_{app}. We assume $\psi{:}\,\mathsf{ctx}$, $m{:}\,\lceil \psi \vdash \mathtt{tm} \rceil$, $n{:}\,\lceil \psi \vdash \mathtt{tm} \rceil$, as well as f_n and f_m which stand for the recursive calls on m and n respectively. We then check that the body t_{app} is well-typed. If we encounter a domain object built with the domain-level constant lam, then we choose the branch b_{lam}. We assume $\psi{:}\,\mathsf{ctx}$ and $m{:}\,\lceil \psi, x{:}\mathtt{tm} \vdash \mathtt{tm} \rceil$ together with the recursive call f_m on m in the extended LF context $\psi, x{:}\mathtt{tm}$. We then check that the body t_{lam} is well-typed. The typing rules for computations are given in Fig. 3. We omit the reduction rules here for brevity.

$\boxed{\Gamma \vdash C : T}$ Contextual object C has contextual type T

$$\frac{\Gamma; \Psi \vdash M : A}{\Gamma \vdash (\widehat{\Psi} \vdash M) : (\Psi \vdash A)} \qquad \frac{\Gamma \vdash \Psi : \mathsf{ctx} \quad x{:}A \in \Psi}{\Gamma \vdash (\widehat{\Psi} \vdash x) : (\Psi \vdash_v A)} \qquad \frac{x{:}\lceil \Phi \vdash_v A \rceil \in \Gamma \quad \Gamma; \Psi \vdash \mathsf{wk}_{\widehat{\Psi}} : \Phi}{\Gamma \vdash (\widehat{\Psi} \vdash \lfloor x \rfloor_{\mathsf{wk}_{\widehat{\Psi}}}) : (\Psi \vdash_v A)}$$

$\boxed{\Gamma \vdash t : \tau}$ Term t has computation type τ

$$\frac{y : \check{\tau} \in \Gamma}{\Gamma \vdash y : \check{\tau}} \qquad \frac{\Gamma \vdash C : T}{\Gamma \vdash \lceil C \rceil : \lceil T \rceil} \qquad \frac{\Gamma \vdash t : (y : \check{\tau}_1) \Rightarrow \tau_2 \quad \Gamma \vdash s : \check{\tau}_1}{\Gamma \vdash t\ s : [s/y]\tau_2} \qquad \frac{\Gamma, y : \check{\tau}_1 \vdash t : \tau_2 \quad \Gamma \vdash (y : \check{\tau}_1) \Rightarrow \tau_2 : \text{ type}}{\Gamma \vdash \mathsf{fn}\ y \Rightarrow t : (y : \check{\tau}_1) \Rightarrow \tau_2}$$

Recursor over domain-level terms $\mathcal{I} = (\psi : \mathsf{ctx}) \Rightarrow (y : \lceil \psi \vdash \mathtt{tm} \rceil) \Rightarrow \tau$

$$\frac{\Gamma \vdash t : \lceil \Psi \vdash \mathtt{tm} \rceil \quad \Gamma \vdash \mathcal{I} : \text{ type} \quad \Gamma \vdash b_v : \mathcal{I} \quad \Gamma \vdash b_{\mathsf{app}} : \mathcal{I} \quad \Gamma \vdash b_{\mathsf{lam}} : \mathcal{I}}{\Gamma \vdash \mathsf{rec}^{\mathcal{I}}\ (b_v \mid b_{\mathsf{app}} \mid b_{\mathsf{lam}})\ \Psi\ t : [\Psi/\psi]\tau}$$

$$\text{Branch for Variable } (b_v) \qquad \frac{\Gamma, \psi : \mathsf{ctx}, p : \lceil \psi \vdash_v \mathtt{tm} \rceil \vdash t_v : \tau}{\Gamma \vdash (\psi, p \mapsto t_v) : \mathcal{I}}$$

$$\text{Branch for Application app } (b_{\mathsf{app}}) \qquad \frac{\Gamma, \psi : \mathsf{ctx}, m{:}\lceil \psi \vdash \mathtt{tm} \rceil, n{:}\lceil \psi \vdash \mathtt{tm} \rceil, f_m{:}\tau, f_n{:}\tau \vdash t_{\mathsf{app}} : \tau}{\Gamma \vdash (\psi, m, n, f_n, f_m \mapsto t_{\mathsf{app}}) : \mathcal{I}}$$

$$\text{Branch for Function lam } (b_{\mathsf{lam}}) \qquad \frac{\Gamma, \phi : \mathsf{ctx}, m{:}\lceil \phi, x{:}\mathtt{tm} \vdash \mathtt{tm} \rceil, f_m{:}[(\phi, x{:}\mathtt{tm})/\psi]\tau \vdash t_{\mathsf{lam}} : [\phi/\psi]\tau}{\Gamma \vdash \psi, m, f_m \mapsto t_{\mathsf{lam}} : \mathcal{I}}$$

Fig. 3. Typing Rules for Contextual Objects and Computations

We now give an interpretation of simply-typed Cocon in a presheaf model with a cartesian closed universe of representables. Let us first extend the internal dependent type theory with the constant $\mathtt{tm}$ for modelling the domain-level type constant $\mathtt{tm}$ and with the constants $\mathsf{app}\colon \mathtt{El}\ \mathtt{tm} \to \mathtt{El}\ \mathtt{tm} \to \mathtt{El}\ \mathtt{tm}$ and $\mathtt{lam}\colon (\mathtt{El}\ \mathtt{tm} \to \mathtt{El}\ \mathtt{tm}) \to \mathtt{El}\ \mathtt{tm}$ to model the corresponding domain-level constants app and lam.

Interpretation of domain-level types

$$\begin{array}{ll}
[\![\mathtt{tm}]\!] & = \mathtt{tm} \\
[\![A \to B]\!] & = \mathtt{arrow}\ [\![A]\!]\ [\![B]\!]
\end{array}$$

Interpretation of domain-level contexts

$$\begin{array}{lll}
[\![\Gamma \vdash \psi : \mathtt{ctx}]\!] & = \psi & \\
[\![\Gamma \vdash \cdot : \mathtt{ctx}]\!] & = \mathtt{unit} & \\
[\![\Gamma \vdash (\Psi, x{:}A) : \mathtt{ctx}]\!] & = \mathtt{times}\ e\ [\![A]\!] & \text{where } [\![\Gamma \vdash \Psi : \mathtt{ctx}]\!] = e
\end{array}$$

Interpretation of domain-level terms, where $[\![\Gamma \vdash \Psi\ \ \mathtt{ctx}]\!] = e$

$$\begin{array}{lll}
[\![\Gamma; \Psi \vdash x : A]\!] & = \curlywedge\, u : \mathtt{El}\ e.\mathtt{snd}\ (\mathtt{fst}^k\ u) & \text{where } \Psi = \Psi_0, x{:}A, y_k{:}A_k, \ldots, y_1{:}A_1 \\
[\![\Gamma; \Psi \vdash \lambda x.\, M : A \to B]\!] & = \curlywedge\, u : \mathtt{El}\ e.\mathtt{arrow\text{-}i}\ (\curlywedge x{:}\mathtt{El}\ [\![A]\!].e'\ (\mathtt{pair}\ u\ x)) & \\
 & & \text{where } [\![\Gamma; \Psi, x{:}A \vdash M : B]\!] = e' \\
[\![\Gamma; \Psi \vdash M\ N : B]\!] & = \curlywedge\, u : \mathtt{El}\ e.\mathtt{arrow\text{-}e}\ (e_1\ u)\ (e_2\ u) & \text{where } [\![\Gamma; \Psi \vdash M : A \to B]\!] = e_1 \\
 & & \text{and } [\![\Gamma; \Psi \vdash N : A]\!] = e_2 \\
[\![\Gamma; \Psi \vdash \lfloor t \rfloor_\sigma : A]\!] & = \mathtt{let\ box}\ x = e_1\ \mathtt{in}\ \curlywedge\, u : \mathtt{El}\ e.x(e_2\ u) & \text{where } [\![\Gamma \vdash t : \lceil \Phi \vdash A \rceil]\!] = e_1 \\
 & & \text{and } [\![\Gamma; \Psi \vdash \sigma : \Phi]\!] = e_2 \\
[\![\Gamma; \Psi \vdash \mathsf{app} : \mathtt{tm} \to \mathtt{tm} \to \mathtt{tm}]\!] & = \curlywedge\, u : \mathtt{El}\ e.\mathtt{arrow\text{-}i}(\curlywedge x{:}\mathtt{El}\ \mathtt{tm}.\ \mathtt{arrow\text{-}i}\ (\curlywedge y{:}\mathtt{El}\ \mathtt{tm}.\ \mathtt{app}\ x\ y)) & \\
[\![\Gamma; \Psi \vdash \mathsf{lam} : (\mathtt{tm} \to \mathtt{tm}) \to \mathtt{tm}]\!] & = \curlywedge\, u : \mathtt{El}\ e.\mathtt{arrow\text{-}i}(\curlywedge f{:}\mathtt{El}\ (\mathtt{arrow}\ \mathtt{tm}\ \mathtt{tm}).\ \mathtt{lam}\ (\curlywedge x{:}\mathtt{El}\ \mathtt{tm}.\ \mathtt{arrow\text{-}e}\ f\ x)) &
\end{array}$$

Interpretation of domain-level substitutions, where $[\![\Gamma \vdash \Psi\ \ \mathtt{ctx}]\!] = e$

$$\begin{array}{lll}
[\![\Gamma; \Psi \vdash \cdot : \cdot]\!] & = \curlywedge\, u : \mathtt{El}\ e.\mathtt{terminal} & \\
[\![\Gamma; \Psi \vdash (\sigma, M) : \Phi, x{:}A]\!] & = \curlywedge\, u : \mathtt{El}\ e.\mathtt{pair}\ (e_1 u)\ (e_2 u) & \\
 & & \text{where } [\![\Gamma; \Psi \vdash \sigma : \Phi]\!] = e_1 \text{ and } [\![\Gamma; \Psi \vdash M : A]\!] = e_2 \\
[\![\Gamma; \Psi, \overrightarrow{x{:}A} \vdash \mathsf{wk}_{\widehat{\Psi}} : \Psi]\!] & = \curlywedge\, u : \mathtt{El}\ e.\mathtt{fst}^n\ u & \text{where } n = |\overrightarrow{x{:}A}|
\end{array}$$

Fig. 4. Interpretation of Domain-level Types and Terms

We can now translate domain-level and computation-level types of Cocon into the internal dependent type theory for $\widehat{\mathbb{D}}$. We do so by interpreting the domain-level terms, types, substitutions, and contexts (see Fig. 4). All translations are on well-typed terms and types. Domain-level types are interpreted as the terms of type $\mathtt{Obj}$ in the internal dependent type theory that represent them. Domain-level contexts are also interpreted as terms of type $\mathtt{Obj}$ by $[\![\Gamma \vdash \Psi : \mathtt{ctx}]\!]$. For example, a domain-level context $x{:}\mathtt{tm}, y{:}\mathtt{tm}$ is interpreted as $\mathtt{times}\ (\mathtt{times}\ \mathtt{unit}\ \mathtt{tm})\ \mathtt{tm} : \mathtt{Obj}$. A domain-level substitution with domain Ψ and codomain Φ becomes a function from $\mathtt{El}\ e'$ to $\mathtt{El}\ e$, where $e' = [\![\Gamma \vdash \Psi : \mathtt{ctx}]\!]$ and $e = [\![\Gamma \vdash \Phi : \mathtt{ctx}]\!]$. Thus we use a semantic function to interpret a simultaneous substitution as usual. As e is some product, for example $\mathtt{times}\ (\mathtt{times}\ \mathtt{unit}\ \mathtt{tm})\ \mathtt{tm}$, the domain-level substitution is translated into a function returning an n-ary tuple. A weakening substitution $\Gamma; \Psi, x{:}\mathtt{tm} \vdash \mathsf{wk}_\Psi : \Psi$ is interpreted as $\mathtt{fst}\ u$ where $u{:}\mathtt{El}\ (\mathtt{times}\ e\ \mathtt{tm})$ and $e = [\![\Gamma \vdash \Psi : \mathtt{ctx}]\!]$. More

Interpretation of contextual objects (C)

$$[\![\Gamma \vdash (\widehat{\Phi} \vdash M) : (\Phi \vdash A)]\!] = [\![\Gamma; \Phi \vdash M : A]\!]$$
$$[\![\Gamma \vdash (\widehat{\Phi} \vdash M) : (\Phi \vdash_v A)]\!] = [\![\Gamma; \Phi \vdash M : A]\!]$$

Interpretation of contextual types (T)

$$[\![\Gamma \vdash (\Phi \vdash A)]\!] = (u{:}\mathtt{El}\ e) \to \mathtt{El}\ [\![A]\!] \quad \text{where } [\![\Gamma \vdash \Phi : \mathtt{ctx}]\!] = e$$
$$[\![\Gamma \vdash (\Phi \vdash_v A)]\!] = (u{:}\mathtt{El}\ e) \to_v \mathtt{El}\ [\![A]\!] \quad \text{where } [\![\Gamma \vdash \Phi : \mathtt{ctx}]\!] = e$$

Fig. 5. Interpretation of Contextual Objects and Types

generally, when we weaken a context Ψ by n declarations, i.e. $\overrightarrow{x{:}A}$, we interpret wk_Ψ as $\mathtt{fst}^n\ u$. A well-typed domain-level term, $\Gamma; \Psi \vdash M : A$, is mapped to a function from $u{:}\mathtt{El}\ [\![\Gamma \vdash \Psi : \mathtt{ctx}]\!]$ to $\mathtt{El}\ [\![A]\!]$.

Hence the translation of a well-typed domain-level term eventually introduces via a $\diagdown u$ that stands for the term-level interpretation of a domain-level context Φ. Most cases in the interpretation just pass u along. The exceptional case is $\Gamma; \Phi \vdash \lambda x.M : A \to B$, because the body M is translated into a function from an extended domain-level context $\Psi, x{:}A$. In this case, we need to pair u with the variable x obtained from the constructor of the domain-level function. When we translate a variable x where $\Phi = \Phi_0, x{:}A, y_k{:}A_k, \ldots, y_1{:}A_1$, we return $\mathsf{snd}\ (\mathtt{fst}^k\ u)$. We translate $\Gamma; \Phi \vdash \lfloor t \rfloor_\sigma : A$ directly using `let box`-construct. Intuitively, a substitution of terms is morphism composition. The interpretation of the domain-level substitution σ has given one morphism. The computation term t should give the other morphism. Since it has the contextual type $\lceil \Phi \vdash \mathtt{tm} \rceil$ its translation will be of type $\flat(\mathtt{El}\ e \to \mathtt{El}\ \mathtt{tm})$ where $e' = [\![\Gamma \vdash \Phi : \mathtt{ctx}]\!]$. We thus can obtain a $\mathtt{El}\ e \to \mathtt{El}\ \mathtt{tm}$ from `let box`, in which we can compose e_2 to obtain the final morphism. The translation of domain-level applications and domain-level constants app and lam is straightforward.

The interpretation of a contextual type $(\Phi \vdash A)$ makes explicit the fact that they correspond to functions $\mathtt{El}\ e \to \mathtt{El}\ [\![A]\!]$ where $e = [\![\Gamma \vdash \Phi : \mathtt{ctx}]\!]$ (see Fig. 5). Consequently, the corresponding contextual object $(\widehat{\Phi} \vdash M)$ is interpreted as a function which is already the case by just taking the interpretation of domain-level terms. The case of $(\Psi \vdash_v A)$ requires the contextual object to be interpreted to the restricted function space denoted by $\to_v$, which is the case by looking at the variable case of the domain-level interpretation.

Last, we give the interpretation of computation-level types, contexts and terms (see Fig. 6). It is mostly straightforward. We simply map $\lceil T \rceil$ in context Γ to $\flat[\![\Gamma \vdash T]\!]$ and $\lceil C \rceil$ is simply interpreted as a boxed term. Since we intend to keep all variables on the computation level crisp, we interpret function types to the crisp function space. As a consequence, the argument in a function application must be closed. This is true because all computation-level terms should only use crisp variables, and it is justified by the soundness theorem. When translating a computation-level function, we use $\diagdown^\flat$ for abstraction, so that we create a crisp function. We then recursively interpret the function body.

The interpretation of the recursor is straightforward now (see Fig. 7). In Lemma 4.5, we expressed a primitive recursion scheme in our internal type theory and defined a term `rec` together with its type. We now interpret every branch of our recursor in the computation-level as a function of the required type in our internal type theory. While this is somewhat tedious, it is straightforward. When interpreting the branches, we again use the crisp function space, which allows us to push the parameters to the crisp context and simulate the behavior of Cocon. The branch body is then interpreted recursively.

Interpretation of computation-level types ($\check{\tau}$)

$$\begin{array}{ll}
[\![\Gamma \vdash \lceil T \rceil]\!] & = \flat [\![\Gamma \vdash T]\!] \\
[\![\Gamma \vdash (x{:}\check{\tau}_1) \Rightarrow \tau_2]\!] & = (x{::}[\![\Gamma \vdash \check{\tau}_1]\!]) \to^{\flat} [\![\Gamma, x : \check{\tau}_1 \vdash \tau_2]\!] \\
[\![\Gamma \vdash \mathtt{ctx}]\!] & = \mathtt{Obj}
\end{array}$$

Computation-level typing contexts (Γ)

$$\begin{array}{ll}
[\![\cdot]\!] & = \cdot \\
[\![\Gamma, x{:}\check{\tau}]\!] & = [\![\Gamma]\!], x :: [\![\Gamma \vdash \check{\tau}]\!]
\end{array}$$

Interpretation of computations ($\Gamma \vdash t : \tau$; without recursor)

$$\begin{array}{lll}
[\![\Gamma \vdash \lceil C \rceil : \lceil T \rceil]\!] & = \mathtt{box}\ e & \text{where } [\![\Gamma \vdash C : T]\!] = e \\
[\![\Gamma \vdash t_1\ t_2 : \tau]\!] & = e_1\ e_2 & \text{where } [\![\Gamma \vdash t_1 : (x{:}\check{\tau}_2) \Rightarrow \tau]\!] = e_1 \\
 & & \text{and } [\![\Gamma \vdash t_2 : \check{\tau}_2]\!] = e_2 \\
[\![\Gamma \vdash \mathsf{fn}\ x \Rightarrow t : (x{:}\check{\tau}_1) \Rightarrow \tau_2]\!] & = \rtimes^{\flat} x{::}[\![\Gamma \vdash \check{\tau}_1]\!].\ e & \text{where } [\![\Gamma, x{:}\check{\tau}_1 \vdash t : \tau_2]\!] = e \\
[\![\Gamma \vdash x : \tau]\!] & = x &
\end{array}$$

Fig. 6. Interpretation of Computation-level Types and Terms – without recursor

Interpretation of recursor for $\mathcal{I} = (\psi : \mathtt{ctx}) \Rightarrow (y : \lceil \psi \vdash \mathtt{tm} \rceil) \Rightarrow \tau$:

$[\![\Gamma \vdash \mathsf{rec}^{\mathcal{I}}\ (b_v \mid b_{\mathsf{app}} \mid b_{\mathsf{lam}})\ \Psi\ t : [\Psi/\psi,\ t/y]\tau]\!] = \mathtt{rec}\ e_v\ e_{\mathsf{app}}\ e_{\mathtt{lam}}\ e_c\ e$
where $[\![\Gamma \vdash b_v : \mathcal{I}]\!] = e_v$, $[\![\Gamma \vdash b_{\mathsf{app}} : \mathcal{I}]\!] = e_{\mathsf{app}}$, $[\![\Gamma \vdash b_{\mathsf{lam}} : \mathcal{I}]\!] = e_{\mathtt{lam}}$,
$[\![\Gamma \vdash \Psi : \mathtt{ctx}]\!] = e_c$ and $[\![\Gamma \vdash t : \lceil \Psi \vdash \mathtt{tm} \rceil]\!] = e$

Interpretation of Variable Branch

$[\![\Gamma \vdash (\psi, p \mapsto t_v) : \mathcal{I}]\!] = \rtimes^{\flat}\psi{::}\mathtt{Obj}.\ \rtimes^{\flat} p{::}\flat(\mathtt{El}\ \psi \to_v \mathtt{El}\ \mathtt{tm}).e$
where $[\![\Gamma, \psi : \mathtt{ctx}, p : \lceil \psi \vdash_v \mathtt{tm} \rceil \vdash t_v : [p/y]\tau]\!] = e$

Interpretation of Application Branch

$[\![\Gamma \vdash (\psi, m, n, f_n, f_m \mapsto t_{\mathsf{app}}) : \mathcal{I}]\!] = \rtimes^{\flat}\psi{::}\mathtt{Obj}.\ \rtimes^{\flat} m, n{::}\flat(\mathtt{El}\ \psi \to \mathtt{El}\ \mathtt{tm}).$
$\rtimes^{\flat} f_m{::}[\![\Gamma \vdash [\psi, m/\psi, y]\tau]\!].\ \rtimes^{\flat} f_n{::}[\![\Gamma \vdash [\psi, n/\psi, y]\tau]\!].e$
where $[\![\Gamma, \psi{:}\mathtt{ctx}, m{:}\lceil \psi \vdash \mathtt{tm} \rceil, n{:}\lceil \psi \vdash \mathtt{tm} \rceil, f_m : [m/y]\tau, f_n : [n/y]\tau \vdash t_{\mathsf{app}} : [\lceil \psi \vdash \mathsf{app}\ \lfloor m \rfloor\ \lfloor n \rfloor \rceil / y]\tau]\!] = e$

Interpretation of Lambda-Abstraction Branch

$[\![\Gamma \vdash (\psi, m, f_m \mapsto t_{\mathsf{lam}}) : \mathcal{I}]\!] = \rtimes^{\flat}\psi :: \mathtt{Obj}.\ \rtimes^{\flat} m :: \flat(\mathtt{El}\ (\mathtt{times}\ \psi\ \mathtt{tm}) \to \mathtt{El}\ \mathtt{tm}).\ \rtimes^{\flat} f_m{::}\tau_m.e$
where $[\![\Gamma \vdash [(\psi, x{:}\mathtt{tm}),\ m/\psi, y]\tau]\!] = \tau_m$,
$[\![\Gamma, \psi{:}\mathtt{ctx}, m{:}\lceil \psi, x{:}\mathtt{tm} \vdash \mathtt{tm} \rceil, f_m : [(\psi, x : tm), m/\psi, y]\tau \vdash t_{\mathsf{app}} : [\lceil \psi \vdash \mathsf{lam}\ \lambda x.\lfloor m \rfloor \rceil / y]\tau]\!] = e$

Fig. 7. Interpretation of Recursor

We can now show that all well-typed domain-level and computation-level objects are translated into well-typed constructions in our internal type theory. As a consequence, we can show that equality in Cocon implies the corresponding equivalence in our internal type theoretic interpretation.

Lemma 5.1. *The interpretation maintains the following typing invariants:*

- *If* $\Gamma \vdash \Psi : \mathtt{ctx}$ *then* $[\![\Gamma \vdash \Psi : \mathtt{ctx}]\!] : \mathtt{Obj}$.
- *If* $\Gamma;\ \Psi \vdash M : A$ *then* $[\![\Gamma]\!] \mid \cdot \vdash [\![\Gamma; \Psi \vdash M : A]\!] : (u : \mathtt{El}\ [\![\Gamma \vdash \Psi : \mathtt{ctx}]\!]) \to \mathtt{El}\ [\![A]\!]$.
- *If* $\Gamma;\ \Psi \vdash \sigma : \Psi$ *then* $[\![\Gamma]\!] \mid \cdot \vdash [\![\Gamma; \Psi \vdash \sigma : \Psi]\!] : (u{:}\mathtt{El}\ [\![\Gamma \vdash \Psi : \mathtt{ctx}]\!]) \to \mathtt{El}\ [\![\Psi]\!]$.
- *If* $\Gamma \vdash C : T$ *then* $[\![\Gamma]\!] \mid \cdot \vdash [\![\Gamma \vdash C : T]\!] : [\![\Gamma \vdash T]\!]$.
- *If* $\Gamma \vdash t : \tau$ *then* $[\![\Gamma]\!] \mid \cdot \vdash [\![\Gamma \vdash t : \tau]\!] : [\![\Gamma \vdash \tau]\!]$.

The proof goes by induction on derivations. Next we show that equivalence in Cocon is preserved by the interpretation.

Proposition 5.2 (Soundness). *The following are true.*

- *If* $\Gamma;\ \Psi \vdash M \equiv N : A$ *then* $[\![\Gamma]\!] \mid \cdot \vdash [\![\Gamma;\ \Psi \vdash M : A]\!] = [\![\Gamma;\ \Psi \vdash N : A]\!] : (u{:}\,\mathtt{El}\ [\![\Psi]\!]) \to \mathtt{El}\ [\![A]\!]$.
- *If* $\Gamma; \Psi \vdash \sigma \equiv \sigma' : \Phi$ *then* $[\![\Gamma]\!] \mid \cdot \vdash [\![\Gamma; \Psi \vdash \sigma : \Phi]\!] = [\![\Gamma; \Psi \vdash \sigma' : \Phi]\!] : (u{:}\,\mathtt{El}\ [\![\Psi]\!]) \to \mathtt{El}\ [\![\Phi]\!]$.
- *If* $\Gamma \vdash t_1 \equiv t_2 : \check{\tau}$ *then* $[\![\Gamma]\!] \mid \cdot \vdash [\![\Gamma \vdash t_1 : \check{\tau}]\!] = [\![\Gamma \vdash t_2 : \check{\tau}]\!] : [\![\Gamma \vdash \check{\tau}]\!]$.

The proof in the β and η equivalence cases of contextual types is interesting. In the case of β equivalence, we have

$$\frac{\Gamma; \Phi \vdash M : A \qquad \Gamma; \Psi \vdash \sigma : \Phi}{\Gamma; \Psi \vdash \lfloor \ulcorner M \urcorner \rfloor_\sigma \equiv [\sigma] M : A}$$

The right hand side is easy; substitution in terms is just composition of morphisms:

$$[\![\Gamma; \Psi \vdash [\sigma]M : A]\!] = e_1 \circ e_2 \qquad \text{where } [\![\Gamma; \Phi \vdash M : A]\!] = e_1 \text{ and } [\![\Gamma; \Psi \vdash \sigma : \Phi]\!] = e_2$$

On the left hand side, we have

$$\begin{aligned} [\![\Gamma; \Psi \vdash \lfloor \ulcorner M \urcorner \rfloor_\sigma : A]\!] &= \mathtt{let\ box}\ x = \mathtt{box}\ e_1\ \mathtt{in}\ \diamond u : \mathtt{El}\ e.x(e_2\ u) && \text{where } [\![\Gamma \vdash \Psi\ \ \mathtt{ctx}]\!] = e, \\ & && [\![\Gamma; \Phi \vdash M : A]\!] = e_1 \text{ and } [\![\Gamma; \Psi \vdash \sigma : \Phi]\!] = e_2 \\ &= \diamond u : \mathtt{El}\ e.e_1(e_2\ u) && \text{due to the } \beta \text{ rule of } \flat \end{aligned}$$

Thus the β equivalence of contextual types is sound semantically.

The following rule expresses the η equivalence of contextual types:

$$\frac{\Gamma \vdash t : \ulcorner \Psi \vdash A \urcorner}{\Gamma \vdash t \equiv \ulcorner \lfloor t \rfloor_{\mathsf{wk}_{\widehat{\Psi}}} \urcorner : \ulcorner \Psi \vdash A \urcorner}$$

We reason as follows:

$$\begin{aligned} [\![\Gamma \vdash \ulcorner \lfloor t \rfloor_{\mathsf{wk}_{\widehat{\Psi}}} \urcorner : \ulcorner \Psi \vdash A \urcorner]\!] &= \mathtt{box\ let\ box}\ x = e'\ \mathtt{in}\ \diamond u : \mathtt{El}\ e.x\ u \\ & && \text{where } [\![\Gamma \vdash \Psi\ \ \mathtt{ctx}]\!] = e \text{ and } [\![\Gamma \vdash t : \ulcorner \Psi \vdash A \urcorner]\!] = e' \\ &= \mathtt{box\ let\ box}\ x = [\![\Gamma \vdash t : \ulcorner \Psi \vdash A \urcorner]\!]\ \mathtt{in}\ x && \eta \text{ equivalence of } \diamond \\ &= [\![\Gamma \vdash t : \ulcorner \Psi \vdash A \urcorner]\!] \end{aligned}$$

The last equation requires a second thought. Indeed, this equation does not normally hold. In fact, box let box $x = t$ in $x = t$ is equivalent to requiring $\flat$ to be idempotent. In our model, this turns out to be true, as we can see from

$$\flat\flat F(\Psi) = \flat F(\top) = F(\top) = \flat F(\Psi)$$

That is, $\flat$ is *definitionally* idempotent, which allows us to conclude the equation. The fact that we rely on the idempotency of the $\flat$ modality implies that the model can only support a two-level modal system.

Domain-level types	A, B	$::= \texttt{ty} \mid \texttt{trm}\, M \mid \Pi x : A.B$
Domain-level terms	M, N	$::= \lambda x.M \mid M\, N \mid x \mid \lfloor t \rfloor_\sigma \mid c$
Domain-level Constants	c	$::= \texttt{o} \mid \texttt{arr} \mid \texttt{lam} \mid \texttt{app}$

Fig. 8. Syntax of dependent Cocon

6 DEPENDENTLY TYPED CASE

In the previous sections, we outlined the interpretation of a simply typed variant of Cocon to a presheaf model. In this section, we demonstrate how the idea can be extended to the dependently typed case. In the dependently typed case, both domain level and computation level have dependent function spaces. This can be used to model intrinsically simply typed languages in the domain level.

```
ty : type.                          trm : ty → type.
o : ty.                             lam : Π a : ty, Π b : ty, (trm a → trm b) → trm (arr a b).

arr : ty → ty → ty.                 app : Π a : ty, Π b : ty, trm (arr a b) → trm a → trm b.
```

In `trm`, we use the type parameter to keep track of the object-level type of an object-level term. We still use HOAS to encode the case of lambda abstraction. As in the simply typed case, we have two distinct function spaces in dependently typed Cocon as well: the weak space that is used in the domain level and is used for HOAS, and the strong space that is used in the computation level and supports induction.

6.1 A Simplified Cocon

To model a dependently typed variant of Cocon capable of encoding this object language, we present the modification to the syntax of the simply typed Cocon in Fig. 8. In the syntax for domain-level types, we add the types for the domain language as well as turn the simple function space into a dependent one. For the terms, we add the corresponding constructors of domain level types, `trm` and `ty`. Compared to Pientka et al. [2019], this version of Cocon is simplified by removing the full hierarchy of universes; as a consequence, types and terms are separated.

The typing rules are shown in Fig. 9 and are changed more significantly. Since domain-level terms can appear in types now, we need to add well-formedness condition for domain-level contexts, $\Gamma \vdash \Psi\ \texttt{ctx}$, and types, $\Gamma; \Psi \vdash A\ \texttt{type}$. In particular, $\texttt{trm}\ M$ is well-formed only if M has type `ty`. In the typing rules, the application rule and the unbox rule shows how dependent types are involved. In particular, in the unbox case, the substitution σ is applied to A as the resulting type. Without loss of generality, we require object-level constructors to be fully applied, e.g. $\texttt{arr}\ a$ is not a valid domain-level term. It helps to simplify the semantic interpretation and allows us to focus on the essential idea of the development.

The computation-level judgments are shown in Fig. 10. Similar to the domain level, we also need well-formedness judgments for computation-level contexts and types. Unlike Cocon defined in Pientka et al. [2019], we need a separate judgment for well-formed types because we do not have universes here. The computation-level language is also dependently typed, as shown in the application rule. We also formulate the induction principle for `trm` and `ty`. The case for `ty` is straightforward as it is a normal algebraic data type. For `trm`, in addition to the cases of `lam` and `app`, we need a case for variables as in the simply typed settings.

$\boxed{\Gamma \vdash \Psi : \mathtt{ctx}}$ The domain-level context Ψ is well-formed

$$\frac{}{\Gamma \vdash \cdot : \mathtt{ctx}} \qquad \frac{\Gamma(\psi) = \mathtt{ctx}}{\Gamma \vdash \psi : \mathtt{ctx}} \qquad \frac{\Gamma \vdash \Phi : \mathtt{ctx} \quad \Gamma; \Psi \vdash A\ \mathtt{type}}{\Gamma \vdash \Phi, x : A : \mathtt{ctx}}$$

From now on, whenever Ψ presents, $\Gamma \vdash \Psi : \mathtt{ctx}$ is assumed.

$\boxed{\Gamma; \Psi \vdash A\ \mathtt{type}}$ The domain-level type A is well-formed

$$\frac{}{\Gamma; \Psi \vdash \mathtt{ty}\ \mathtt{type}} \qquad \frac{\Gamma; \Psi \vdash M : \mathtt{ty}}{\Gamma; \Psi \vdash \mathtt{trm}\ M\ \mathtt{type}} \qquad \frac{\Gamma; \Psi \vdash A\ \mathtt{type} \qquad \Gamma; \Psi, x : A \vdash B\ \mathtt{type}}{\Gamma; \Psi \vdash \Pi x : A.B\ \mathtt{type}}$$

$\boxed{\Gamma; \Psi \vdash M : A}$ Term M has type A in domain-level context Ψ and context Γ

$$\frac{x : A \in \Psi}{\Gamma; \Psi \vdash x : A} \qquad \frac{\Gamma; \Psi, x : A \vdash M : B}{\Gamma; \Psi \vdash \lambda x.M : \Pi x : A.B} \qquad \frac{\Gamma; \Psi \vdash M : \Pi x : A.B \qquad \Gamma; \Psi \vdash N : A}{\Gamma; \Psi \vdash M\ N : [N/x]B}$$

$$\frac{\Gamma \vdash t : \lceil \Phi \vdash A \rceil \text{ or } \Gamma \vdash t : \lceil \Phi \vdash_v A \rceil \qquad \Gamma; \Psi \vdash \sigma : \Phi}{\Gamma; \Psi \vdash \lfloor t \rfloor_\sigma : [\sigma]A} \qquad \frac{\Gamma; \Psi \vdash M : A \qquad \Gamma; \Psi \vdash A \equiv B\ \mathtt{type}}{\Gamma; \Psi \vdash M : B} \qquad \frac{}{\Gamma; \Psi \vdash \mathsf{o} : \mathtt{ty}}$$

$$\frac{\Gamma; \Psi \vdash a : \mathtt{ty} \qquad \Gamma; \Psi \vdash b : \mathtt{ty}}{\Gamma; \Psi \vdash \mathtt{arr}\ a\ b : \mathtt{ty}} \qquad \frac{\Gamma; \Psi \vdash a : \mathtt{ty} \qquad \Gamma; \Psi \vdash b : \mathtt{ty} \qquad \Gamma; \Psi \vdash f : \mathtt{trm}\ a \to \mathtt{trm}\ b}{\Gamma; \Psi \vdash \mathtt{lam}\ a\ b\ f : \mathtt{trm}\ (\mathtt{arr}\ a\ b)}$$

$$\frac{\Gamma; \Psi \vdash a : \mathtt{ty} \qquad \Gamma; \Psi \vdash b : \mathtt{ty} \qquad \Gamma; \Psi \vdash m : \mathtt{trm}\ (\mathtt{arr}\ a\ b) \qquad \Gamma; \Psi \vdash n : \mathtt{trm}\ a}{\Gamma; \Psi \vdash \mathtt{app}\ a\ b\ m\ n : \mathtt{trm}\ b}$$

$\boxed{\Gamma; \Psi \vdash \sigma : \Phi}$ Substitution σ provides a mapping from the (domain) context Φ to Ψ

$$\frac{\Gamma \vdash \Psi\ \mathtt{ctx}}{\Gamma; \Psi \vdash \cdot : \cdot} \qquad \frac{\Gamma; \Psi \vdash \sigma : \Phi \qquad \Gamma; \Psi \vdash M : [\sigma]A}{\Gamma; \Psi \vdash \sigma, M : \Phi, x : A} \qquad \frac{\Gamma \vdash \Psi, \overrightarrow{x : A}\ \mathtt{ctx}}{\Gamma; \Psi, \overrightarrow{x : A} \vdash \mathsf{wk}_{\widehat{\Psi}} : \Psi}$$

Fig. 9. Domain-level judgments

6.2 Categories with Families

In the previous section, we showed that we can regard the domain-level language as a cartesian closed category $\mathbb{D}$ and use the Yoneda embedding to embed the domain level into the presheaf category $\widehat{\mathbb{D}}$, which is regarded as the computation-level language. Interestingly, this model can be extended to the case of dependently typed domain languages. In this case, $\mathbb{D}$ needs to have enough structure to model dependent types, and the model we consider here is categories with families (CwFs) [Dybjer 1995; Hofmann 1997].

Definition 6.1. A *category with families* C consists of the following data:

(1) a terminal object $\top$,
(2) a functor $\mathsf{Ty} : C^{op} \to \mathcal{S}et$, whose action on morphisms we denote as $-\{\sigma\} : \mathsf{Ty}(\Phi) \to \mathsf{Ty}(\Psi)$ for $\sigma : \Psi \to \Phi$,
(3) for $\Phi \in C$ and $A \in \mathsf{Ty}(\Phi)$, a set $\mathsf{Tm}(\Phi, A)$, such that:
   - for $\sigma : \Psi \to \Phi$ and $t \in \mathsf{Tm}(\Phi, A)$, $t\{\sigma\} \in \mathsf{Tm}(\Psi, A\{\sigma\})$, and

$\boxed{\vdash \Gamma}$ Γ is a well-formed context

$$\frac{}{\vdash \cdot} \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash \check{\tau}\ \ \mathtt{type}}{\vdash \Gamma, x : \check{\tau}}$$

$\boxed{\Gamma \vdash \check{\tau}\ \ \mathtt{type}}$ $\check{\tau}$ is a well-formed type in context Γ

$$\frac{}{\Gamma \vdash \mathtt{ctx}\ \ \mathtt{type}} \qquad \frac{\Gamma \vdash \check{\tau}_1\ \ \mathtt{type} \qquad \Gamma, y : \check{\tau}_1 \vdash \tau_2\ \ \mathtt{type}}{\Gamma \vdash (y : \check{\tau}_1) \Rightarrow \tau_2\ \ \mathtt{type}} \qquad \frac{\Gamma; \Psi \vdash A\ \ \mathtt{type}}{\Gamma \vdash \lceil \Psi \vdash A \rceil\ \ \mathtt{type}} \qquad \frac{\Gamma; \Psi \vdash A\ \ \mathtt{type}}{\Gamma \vdash \lceil \Psi \vdash_v A \rceil\ \ \mathtt{type}}$$

$\boxed{\Gamma \vdash C : T}$ Contextual object C has contextual type T

$$\frac{}{\Gamma \vdash (\widehat{\Psi} \vdash M) : (\Psi \vdash A)} \qquad \frac{x : A \in \Psi}{\Gamma \vdash (\widehat{\Psi} \vdash x) : (\Psi \vdash_v A)} \qquad \frac{x : \lceil \Phi \vdash_v A \rceil \in \Gamma \qquad \Gamma; \Psi \vdash \mathsf{wk}_{\widehat{\Psi}} : \Phi}{\Gamma \vdash (\widehat{\Psi} \vdash \lfloor x \rfloor_{\mathsf{wk}_{\widehat{\Psi}}}) : (\Psi \vdash_v A)}$$

$\boxed{\Gamma \vdash t : \check{\tau}}$ Term t has computation type $\check{\tau}$

$$\frac{y : \check{\tau} \in \Gamma}{\Gamma \vdash y : \check{\tau}} \qquad \frac{\Gamma \vdash C : T}{\Gamma \vdash \lceil C \rceil : \lceil T \rceil} \qquad \frac{\Gamma, y : \check{\tau}_1 \vdash t : \tau_2}{\Gamma \vdash \mathtt{fn}\ y \Rightarrow t : (y : \check{\tau}_1) \Rightarrow \tau_2}$$

$$\frac{\Gamma \vdash t : (y : \check{\tau}_1) \Rightarrow \tau_2 \qquad \Gamma \vdash s : \check{\tau}_1}{\Gamma \vdash t\ s : [s/y]\tau_2} \qquad \frac{\Gamma \vdash t : \check{\tau}_1 \qquad \Gamma \vdash \check{\tau}_1 \equiv \check{\tau}_2\ \ \mathtt{type}}{\Gamma \vdash t : \check{\tau}_2}$$

Recursor over ty: $\mathcal{I} = (\psi : \mathtt{ctx}) \Rightarrow (y : \lceil \psi \vdash \mathtt{ty} \rceil) \Rightarrow \tau$

$$\frac{\Gamma \vdash t : \lceil \Psi \vdash \mathtt{ty} \rceil \qquad \Gamma \vdash b_{\mathsf{o}} : \mathcal{I} \qquad \Gamma \vdash b_{\mathtt{arr}} : \mathcal{I}}{\Gamma \vdash \mathtt{rec}^{\mathcal{I}}\ (b_{\mathsf{o}} \mid b_{\mathtt{arr}})\ \Psi\ t : [\Psi, t/\psi, y]\tau}$$

Branch for the o case $$\frac{\Gamma, \psi : \mathtt{ctx} \vdash t_{\mathsf{o}} : [\lceil \psi \vdash \mathsf{o} \rceil / y]\tau}{\Gamma \vdash (\psi \mapsto t_{\mathsf{o}}) : \mathcal{I}}$$

Branch for the arr case $$\frac{\Gamma, \psi : \mathtt{ctx}, m, n : \lceil \psi \vdash \mathtt{ty} \rceil, f_m : [m/y]\tau, f_n : [n/y]\tau \vdash t_{arr} : [\lceil \psi \vdash \mathtt{arr} \lfloor m \rfloor \lfloor n \rfloor \rceil / y]\tau}{\Gamma \vdash (\psi, m, n, f_m, f_n \mapsto t_{\mathtt{arr}}) : \mathcal{I}}$$

Recursor over trm: $\mathcal{I} = (\psi : \mathtt{ctx}) \Rightarrow (z : \lceil \vdash \mathtt{ty} \rceil) \Rightarrow (y : \lceil \psi \vdash \mathtt{trm}\ \lfloor z \rfloor . \rceil) \Rightarrow \tau$

$$\frac{\Gamma \vdash t : \lceil \vdash \mathtt{ty} \rceil \qquad \Gamma \vdash t' : \lceil \Psi \vdash \mathtt{trm}[\lfloor t \rfloor .] \rceil \qquad \Gamma \vdash b_v : \mathcal{I} \qquad \Gamma \vdash b_{\mathtt{lam}} : \mathcal{I} \qquad \Gamma \vdash b_{\mathtt{app}} : \mathcal{I}}{\Gamma \vdash \mathtt{rec}^{\mathcal{I}}\ (b_v \mid b_{\mathtt{lam}} \mid b_{\mathtt{app}})\ \Psi\ t\ t' : [\Psi, t, t'/\psi, z, y]\tau}$$

Branch for the variable case $$\frac{\Gamma, \psi : \mathtt{ctx}, a : \lceil \vdash \mathtt{ty} \rceil, t : \lceil \psi \vdash_v \mathtt{trm}\ \lfloor a \rfloor . \rceil \vdash t_v : [a, t/z, y]\tau}{\Gamma \vdash (\psi, a, t \mapsto t_v) : \mathcal{I}}$$

Branch for the lam case $$\frac{\begin{array}{c}\Gamma, \psi : \mathtt{ctx}, a, b : \lceil \vdash \mathtt{ty} \rceil, m : \lceil \psi, x : \mathtt{trm}\ \lfloor a \rfloor . \vdash \mathtt{trm}\ \lfloor b \rfloor . \rceil, \\ f_m : [(\psi, \mathtt{trm}\ \lfloor a \rfloor .), b, m/\psi, x, y]\tau \vdash t_{\mathtt{lam}} : [\lceil \vdash \mathtt{arr} \lfloor a \rfloor \lfloor b \rfloor \rceil, \lceil \psi \vdash \mathtt{lam} \lfloor a \rfloor . \lfloor b \rfloor . (\lambda x . \lfloor m \rfloor) \rceil / z, y]\tau\end{array}}{\Gamma \vdash (\psi, a, b, m, f_m \mapsto t_{\mathtt{lam}}) : \mathcal{I}}$$

Branch for the app case $$\frac{\begin{array}{c}\Gamma, \psi : \mathtt{ctx}, a, b : \lceil \vdash \mathtt{ty} \rceil, m : \lceil \psi \vdash \mathtt{trm}(\mathtt{arr} \lfloor a \rfloor . \lfloor b \rfloor .) \rceil, n : \lceil \psi \vdash \mathtt{trm} \lfloor a \rfloor . \rceil, \\ f_m : [\lceil \vdash \mathtt{arr} \lfloor a \rfloor \lfloor b \rfloor \rceil, m/z, y]\tau, f_n : [a, n/z, y]\tau \vdash t_{\mathtt{app}} : [b, \lceil \psi \vdash \mathtt{app} \lfloor a \rfloor . \lfloor b \rfloor . \lfloor m \rfloor \lfloor n \rfloor \rceil / z, y]\tau\end{array}}{\Gamma \vdash (\psi, a, b, m, n, f_m, f_n \mapsto t_{\mathtt{app}}) : \mathcal{I}}$$

Fig. 10. Judgments in the computation level

- the equations $t\{id_\Phi\} = t$ and $t\{\sigma\}\{\delta\} = t\{\sigma \circ \delta\}$ are valid.

(4) a context comprehension $-.-$ so that given $\Phi \in \mathcal{C}$ and $A \in \mathsf{Ty}(\Phi)$, $\Phi.A \in \mathcal{C}$,
(5) for $\Phi \in \mathcal{C}$ and $A \in \mathsf{Ty}(\Phi)$, a projection morphism of context comprehension $p(A) : \Phi.A \to \Phi$,
(6) for $\Phi \in \mathcal{C}$ and $A \in \mathsf{Ty}(\Phi)$, a variable projection $v_A \in \mathsf{Tm}(\Phi.A, A\{p(A)\})$, and
(7) for $\sigma : \Psi \to \Phi$, $A \in \mathsf{Ty}(\Phi)$, and $t \in \mathsf{Tm}(\Psi, A\{\sigma\})$, a unique extension morphism $\langle \sigma, t \rangle : \Psi \to \Phi.A$.

The following equations hold:

(1) $p(A) \circ \langle \sigma, t \rangle = \sigma$,
(2) $v_A\{\langle \sigma, t \rangle\} = t$, and
(3) $\sigma = \langle p(A) \circ \sigma, v_T\{\sigma\} \rangle$ where $\sigma : \Phi \to \Psi.A$.

We regard the objects in $\mathcal{C}$ as contexts and the morphisms as substitutions of one context for another. Based on this understanding, $\mathsf{Ty}(\Phi)$ denotes the set of semantic types in context Φ. Given a semantic type $A \in \mathsf{Ty}(\Phi)$, $A\{\sigma\}$ is thus applying the substitution σ to A given $\sigma : \Psi \to \Phi$. Based on their definitions, we can prove the following properties of type and term substitutions. Given $A \in \mathsf{Ty}(\Phi)$, $\sigma : \Psi \to \Phi$, $\delta : \Psi' \to \Psi$, $t : \mathsf{Tm}(\Phi, A)$:

$$A\{id_\Phi\} = A$$
$$A\{\sigma\}\{\delta\} = A\{\sigma \circ \delta\}$$

$\mathsf{Tm}(\Phi, A)$ denotes the set of semantic terms of type A in context Φ. Given a term $t \in \mathsf{Tm}(\Phi, A)$ and a substitution $\sigma : \Psi \to \Phi$, $t\{\sigma\} : \mathsf{Tm}(\Psi, A\{\sigma\})$ is the result of applying σ to t. Note that the type of this term is $A\{\sigma\}$, so CwFs are capable of handling dependent types.

Sometimes, given a substitution $\sigma : \Psi \to \Phi$ and $A \in \mathsf{Ty}(\Phi)$, we would like to obtain another substitution $q(\sigma, A) : \Psi.A\{\sigma\} \to \Phi.A$. This substitution is needed below when we define substitution for Π types. We can define

$$q(\sigma, A) := \langle \sigma \circ p(A\{\sigma\}), v_{A\{\sigma\}} \rangle$$

By applying the property of $p(A\{\sigma\})$, we can see that the following diagram is a pullback:

$$\begin{array}{ccc} \Psi.A\{\sigma\} & \xrightarrow{q(\sigma,A)} & \Phi.A \\ {\scriptstyle p(A\{\sigma\})}\downarrow & & \downarrow{\scriptstyle p(A)} \\ \Psi & \xrightarrow[\sigma]{} & \Phi \end{array}$$

We shall work with telescopes of types. A *telescope* of types in context Φ is a sequence of types $A_1, A_2, \ldots, A_n$ such that $A_1 \in \mathsf{Ty}(\Phi)$, $A_2 \in \mathsf{Ty}(\Phi.A_1)$, $\ldots$, $A_n \in \mathsf{Ty}(\Phi.A_1.\ldots.A_{n-1})$. We write $\vec{A}$ to range over telescopes and extend context comprehension, substitution and projection to telescopes in the canonical way. That is, we write $\Phi.\vec{A}$ for $\Phi.A_1.\ldots.A_n$ and $p(\vec{A})$ for $p(A_1) \circ \cdots \circ p(A_n)$.

Up until this point, we have obtained a generic categorical structure for dependent type theory. In order to model the dependent function space, we need a semantic type former.

Definition 6.2. [Hofmann 1997] Semantic Π types in a CwF $\mathcal{C}$ have the following data:

(1) A semantic type $\Pi(A, B) \in \mathsf{Ty}(\Phi)$ for each $A \in \mathsf{Ty}(\Phi)$ and $B \in \mathsf{Ty}(\Phi.A)$,
(2) a semantic term $\Lambda_{A,B}(M) \in \mathsf{Tm}(\Phi, \Pi(A, B))$ for each $M \in \mathsf{Tm}(\Phi.A, B)$, and
(3) a semantic term $\mathsf{App}_{A,B}(M, N) \in \mathsf{Tm}(\Phi, B\{\langle id_\Phi, N \rangle\})$ for each $M \in \mathsf{Tm}(\Phi, \Pi(A, B))$ and $N \in \mathsf{Tm}(\Phi, A)$.

so that the following axioms are satisfied:

(1) $\Pi(A, B)\{\sigma\} = \Pi(A\{\sigma\}, B\{q(\sigma, A)\}) \in \mathsf{Ty}(\Psi)$ for $\sigma : \Psi \to \Phi$,

(2) $\Lambda_{A,B}(M)\{\sigma\} = \Lambda_{A\{\sigma\},B\{q(\sigma,A)\}}(M\{q(\sigma,A)\}) \in \mathsf{Tm}(\Psi, \Pi(A,B)\{\sigma\})$ for $M \in \mathsf{Tm}(\Phi.A, B)$ and $\sigma : \Psi \to \Phi$,
(3) $\mathsf{App}_{A,B}(M,N)\{\sigma\} = \mathsf{App}_{A\{\sigma\},B\{q(\sigma,A)\}}(M\{\sigma\}, N\{\sigma\}) \in \mathsf{Tm}(\Psi, B\{\langle id_\Phi, N\rangle\}\{\sigma\}) = \mathsf{Tm}(\Psi, B\{\langle \sigma, N\{\sigma\}\rangle\})$ for $M \in \mathsf{Tm}(\Phi, \Pi(A,B))$, $N \in \mathsf{Tm}(\Phi, A)$ and $\sigma : \Psi \to \Phi$,
(4) $\mathsf{App}_{A,B}(\Lambda_{A,B}(M), N) = M\{\langle id_\Phi, N\rangle\} \in \mathsf{Tm}(\Phi, B\{\langle id_\Phi, N\rangle\})$ for each $M \in \mathsf{Tm}(\Phi.A, B)$ and $N \in \mathsf{Tm}(\Phi, A)$, and
(5) $\Lambda_{A,B}(\mathsf{App}_{A,B}(M\{p(A)\}, v_A)) = M \in \mathsf{Tm}(\Phi, \Pi(A,B))$.

We often omit the subscripts of Λ and App in favor of conciseness when they can be unambiguously inferred. For example, the fourth axiom can be more concisely expressed as $\mathsf{App}(\Lambda(M), N) = M\{\langle id_\Phi, N\rangle\}$.

One characteristic of this framework is that the consistency of equations involves reasoning about equality between sets. Consider the third equation above. We can transform the left hand side as follows:

$$\begin{aligned}
\mathsf{App}(M,N)\{\sigma\} &\in \mathsf{Tm}(\Phi, B\{\langle id_\Psi, N\rangle\}\{\sigma\}) \\
&= \mathsf{Tm}(\Phi, B\{\langle id_\Psi, N\rangle \circ \sigma\}) \\
&= \mathsf{Tm}(\Phi, B\{\langle \sigma, N\{\sigma\}\rangle\}) && \text{property of extension morphism}
\end{aligned}$$

The right hand side has:

$$\begin{aligned}
\mathsf{App}(M\{\sigma\}, N\{\sigma\}) &\in \mathsf{Tm}(\Phi, B\{q(\sigma,A)\}\{\langle id_\Phi, N\{\sigma\}\rangle\}) \\
&= \mathsf{Tm}(\Phi, B\{q(\sigma,A) \circ \langle id_\Phi, N\{\sigma\}\rangle\}) \\
&= \mathsf{Tm}(\Phi, B\{\langle \sigma \circ p(A\{\sigma\}), v_{A\{\sigma\}}\rangle \circ \langle id_\Phi, N\{\sigma\}\rangle\}) \\
&= \mathsf{Tm}(\Phi, B\{\langle \sigma, N\{\sigma\}\rangle\})
\end{aligned}$$

As both terms belong to the same set, the equation is well-defined.

Combining the definition of a CwF with Π types, we are able to capture the nature of dependent types with dependent function spaces in both the domain level and the computation level. We can also axiomatize other types like `ty` and `trm` and their constructors. However, we need more to interpret Cocon:

(1) CwFs do not provide a direct connection between the domain and the computation level;
(2) the domain-level terms exhibit different substitution behavior from the computation terms.

To overcome these issues, we can extend the model presented in Section 2 by requiring the domain category to possess structure of a CwF. We will expand our discussion in the next section.

6.3 Presheaves over a Small Category with Families

Having introduced an appropriate notion of model for dependent type theories, we can now generalise the construction from Section 4 to the case of dependent domain languages.

As in the simply-typed case, we begin with a term model $\mathbb{D}$ of the domain-level type theory. The main difference is that in the dependent case, this category has the structure of a CwF instead of being just cartesian closed. In order to interpret the computation types of Cocon, we work in the presheaf category $\widehat{\mathbb{D}}$. This category has enough structure to interpret Cocon computations and also embeds $\mathbb{D}$ fully and faithfully via the Yoneda embedding.

A Universe for a small CwF. For working with the internal type theory of $\widehat{\mathbb{D}}$, it is again convenient to capture the embedding of $\mathbb{D}$ into $\widehat{\mathbb{D}}$ in terms of a Tarski-style universe. It is given by the following term and type constants in the internal type theory of $\widehat{\mathbb{D}}$. The types `Ctx` and `El` Φ generalize `Obj`

and $\mathtt{El}\ a$ from Section 4 to the dependent case:

$$\cdot \vdash \mathsf{Ctx}\ \ \mathsf{type} \qquad \Phi : \mathsf{Ctx} \vdash \mathtt{El}\,(\Phi)\ \ \mathsf{type} \qquad \Phi : \mathsf{Ctx} \vdash \mathsf{Ty}(\Phi)\ \ \mathsf{type}$$

$$\Phi : \mathsf{Ctx},\ A : \mathsf{Ty}(\Phi) \vdash \mathtt{Tm}(\Phi, A)\ \ \mathsf{type} \qquad \cdot \vdash \top : \mathsf{Ctx} \qquad \Phi : \mathsf{Ctx},\ A : \mathsf{Ty}(\Phi) \vdash \Phi.A : \mathsf{Ctx}$$

$$\Phi : \mathsf{Ctx} \vdash ! : \mathtt{El}\,(\Phi) \to \mathtt{El}\,(\top) \qquad \Phi : \mathsf{Ctx},\ A : \mathsf{Ty}(\Phi) \vdash p : \mathtt{El}\,(\Phi.A) \to \mathtt{El}\,(\Phi)$$

$$\Phi : \mathsf{Ctx},\ A : \mathsf{Ty}(\Phi) \vdash v : \mathtt{Tm}(\Phi.A, A\{p\})$$

$$\Phi : \mathsf{Ctx},\ \Psi : \mathsf{Ctx},\ A : \mathsf{Ty}(\Psi),\ \sigma : \mathtt{El}\,(\Phi) \to \mathtt{El}\,(\Psi) \vdash A\{\sigma\} : \mathsf{Ty}(\Phi)$$

$$\Phi, \Psi : \mathsf{Ctx},\ A : \mathsf{Ty}(\Psi),\ M : \mathtt{Tm}(\Psi, A),\ \sigma : \mathtt{El}\,(\Phi) \to \mathtt{El}\,(\Psi) \vdash M\{\sigma\} : \mathtt{Tm}(\Phi, A\{\sigma\})$$

$$\Psi, \Phi : \mathsf{Ctx},\ \sigma : \mathtt{El}\,(\Psi) \to \mathtt{El}\,(\Phi),\ A : \mathsf{Ty}(\Phi),\ M : \mathtt{Tm}(\Psi, A\{\sigma\}) \vdash \langle \sigma, M\rangle : \mathtt{El}\,(\Psi) \to \mathtt{El}\,(\Phi.A)$$

Let us outline the interpretation of these constants in $\widehat{\mathbb{D}}$ next. Recall from Section 3 that contexts are objects of $\widehat{\mathbb{D}}$, types in context Γ are presheaves $\widehat{\mathsf{Ty}}(\Gamma) = \int \Gamma^{\mathrm{op}} \to \mathit{Set}$, and terms are sections of the projections maps $p\colon \Gamma.A \to \Gamma$ in $\widehat{\mathbb{D}}$. We detail the required structure used in the interpretation.

The interpretation of the types Ctx and $\mathtt{El}\,(\Phi)$ is as follows:

$$\mathsf{Ctx} : \widehat{\mathsf{Ty}}(\top)$$

$$\mathsf{Ctx}(\Psi, *) = \{\vec{A} \mid \vec{A} \text{ is a telescope of types in context } \Psi\}$$

$$\mathsf{Ctx}(\sigma : (\Psi', *) \to (\Psi, *)) = \vec{A} \mapsto \vec{A}\{\sigma\}$$

$$\mathtt{El}\ : \widehat{\mathsf{Ty}}(\top.\mathsf{Ctx})$$

$$\mathtt{El}\,(\Psi, (*, \vec{A})) = \{\sigma \in \mathbb{D}(\Psi, \Psi.\vec{A}) \mid p(\vec{A}) \circ \sigma = \mathsf{id}_\Psi\}$$

$$\mathtt{El}\,(\sigma : (\Psi', (*, \vec{A}\{\sigma\})) \to (\Psi, (*, \vec{A}))) = f \in \mathbb{D}(\Psi, \Psi.\vec{A}) \mapsto f\{\sigma\}$$

The types of the form $\mathsf{Ty}(\Phi)$ in the internal type theory are interpreted as follows:

$$\mathsf{Ty} : \widehat{\mathsf{Ty}}(\top.\mathsf{Ctx})$$

$$\mathsf{Ty}(\Psi, (*, \vec{A})) = \mathsf{Ty}(\Psi.\vec{A})$$

$$\mathsf{Ty}(\sigma : (\Psi', (*, \vec{A}\{\sigma\}) \to (\Psi, (*, \vec{A}))) = B \in \mathsf{Ty}(\Psi.\vec{A}) \mapsto B\{q(\sigma, \vec{A})\}$$

Here Ty on the right hand side is given by the CwF structure of the domain category $\mathbb{D}$. That is, Ty is defined in terms of Ty in the domain category, so all types of the domain level can be referred to as terms in the presheaf category.

The types of the form $\mathtt{Tm}(\Phi, A)$ are interpreted as follows.

$$\mathtt{Tm} : \widehat{\mathsf{Ty}}(\top.\mathsf{Ctx}.\mathsf{Ty})$$

$$\mathtt{Tm}(\Psi, (*, \vec{A}, B)) = \mathsf{Tm}(\Psi.\vec{A}, B)$$

$$\mathtt{Tm}(\sigma) = M \mapsto M\{q(\sigma, \vec{A})\}$$

Similar to Ty, Tm is also defined by terms Tm in the domain category $\mathbb{D}$.

For spelling out the interpretation of the remaining terms, we need to give a manageable presentation of the semantic interpretation of function types of the form $\mathtt{El}\,(\Phi) \to \mathtt{El}\,(\Phi')$. In the simply typed case, the Yoneda lemma shows an isomorphism between $\mathbb{D}(\Phi, \Phi')$ and $\widehat{\mathbb{D}}(\mathrm{y}(\Phi), \mathrm{y}(\Phi'))$. In the dependently typed case, this isomorphism remains: the function space $\mathtt{El}\,(\Phi) \to \mathtt{El}\,(\Phi')$ is isomorphic to $\mathbb{D}(\Phi, \Phi')$, as proved by the following lemma:

Lemma 6.3. *The interpretation of*

$$\Phi\colon \mathsf{Ctx}, \Phi'\colon \mathsf{Ctx} \vdash \mathtt{El}\,(\Phi) \to \mathtt{El}\,(\Phi')\ \ \mathsf{type}$$

in the internal type theory of $\widehat{\mathbb{D}}$ *is isomorphic to the following presheaf Hom:*

$$\mathit{Hom}\colon \widehat{\mathsf{Ty}}(\top.\mathsf{Ctx}.\mathsf{Ctx}\{p\})$$

$$\mathit{Hom}(\Psi, (*, \vec{A}, \vec{B})) = \{\sigma \in \mathbb{D}(\Psi.\vec{A}, \Psi.\vec{B}) \mid p(\vec{B}) \circ \sigma = p(\vec{A})\}$$

Proof. For one direction, as given in Section 3, the type $\mathtt{El}\,(\Phi) \to \mathtt{El}\,(\Phi')$ in the syntax is interpreted to a function f which takes $\sigma\colon \Phi \to \Psi$ and $M : \Phi \to \Phi.\vec{A}\{\sigma\}$ and outputs $\Phi \to \Phi.\vec{B}\{\sigma\}$ that is natural in Φ. Then we obtain $f(p_{\vec{A}} : \Psi.\vec{A} \to \Psi, v : \Psi.\vec{A} \to \Psi.\vec{A}.\vec{A}\{p_{\vec{A}}\}) : \Psi.\vec{A} \to \Psi.\vec{A}.\vec{B}\{p_{\vec{A}}\}$, such that $p_{\vec{A}} \circ f(p_{\vec{A}}, v) = \mathsf{id}_{\Psi.\vec{A}}$. We obtain $\Psi.\vec{A} \to \Psi.\vec{B}$ by composing $q(p_{\vec{A}}, \vec{B})$ with $f(p_{\vec{A}}, v)$.

The other direction is given $\sigma : \Psi.\vec{A} \to \Psi.\vec{B}$, $\delta : \Phi \to \Psi$ and $\sigma' : \Phi \to \Phi.\vec{A}\{\delta\}$ and outputs $\Phi \to \Phi.\vec{B}\{\delta\}$.

$$\begin{array}{ccccc} & & \Psi.\vec{A} & \xrightarrow{\ \sigma\ } & \Psi.\vec{B} \\ & & {\scriptstyle q(\delta,\vec{A})}\uparrow & & \uparrow{\scriptstyle q(\delta,\vec{B})} \\ \Phi & \xrightarrow{\ \sigma'\ } & \Phi.\vec{A}\{\delta\} & \longrightarrow & \Phi.\vec{B}\{\delta\} \end{array}$$

Thus the morphism $\Phi \to \Phi.\vec{B}\{\delta\}$ is obtained by composing σ' with the unique morphism that makes the above square a pullback. □

In the following, we shall apply the isomorphism from the lemma implicitly and treat the interpretation of $\mathtt{El}\,(\Phi) \to \mathtt{El}\,(\Phi')$ to be the same as Hom.

With this convention, we are able to give an explicit interpretation for the judgments that we give at the beginning of this section. Recall, for example, the term for type substitution:

$$\Phi : \mathsf{Ctx},\ \Psi : \mathsf{Ctx},\ A : \mathsf{Ty}(\Psi),\ \sigma : \mathtt{El}\,(\Phi) \to \mathtt{El}\,(\Psi) \vdash A\{\sigma\} : \mathsf{Ty}(\Phi)$$

This term $A\{\sigma\}$ is interpreted as the following natural transformation:

$$A\{\sigma\}_\Theta := (*, \vec{A}, \vec{B}, C : \mathsf{Ty}(\Theta.\vec{B}), \sigma : \Theta.\vec{A} \to \Theta.\vec{B}) \mapsto (*, \vec{A}, \vec{B}, C, \sigma, C\{\sigma\})$$

Recall that terms of $\widehat{\mathbb{D}}$ are interpreted as sections of context projections, so $A\{\sigma\}$ is a natural transformation from the interpretation of the context $\Gamma := \Phi : \mathsf{Ctx}, \Psi : \mathsf{Ctx}, A : \mathsf{Ty}(\Psi), \sigma : \mathtt{El}\,(\Phi) \to \mathtt{El}\,(\Psi)$ to the interpretation of the context $\Gamma, B : \mathsf{Ty}(\Phi)$. $\vec{A}$ and $\vec{B}$ interpret Φ and Ψ respectively because Ctx is interpreted as a set of telescopes of Θ.

We omit the interpretation of the remaining terms and the verification of the equations and next turn to the case where $\mathbb{D}$ is a CwF with dependent product.

Domain-level Π *types.* Domain-level Π types can be formulated by the following constants:

$$\Phi : \mathsf{Ctx}, A : \mathsf{Ty}(\Phi), B : \mathsf{Ty}(\Phi.A) \vdash \Pi(A, B) : \mathsf{Ty}(\Phi)$$

$$\Phi : \mathsf{Ctx}, M : \mathtt{Tm}(\Phi.A, B) \vdash \Lambda(M) : \mathtt{Tm}(\Phi, \Pi(A, B))$$

$$\Phi : \mathsf{Ctx}, M : \mathtt{Tm}(\Phi, \Pi(A, B)), N : \mathtt{Tm}(\Phi, A) \vdash \mathsf{App}(M, N) : \mathtt{Tm}(\Phi, B\{\langle 1_\Phi, N\rangle\})$$

Π types defined here reside in the presheaf category and model Π types in the domain category $\mathbb{D}$. They can be defined in terms of Π types in $\mathbb{D}$ as the types defined above:

$$\Pi_\Psi := (*, \vec{A}, C : \mathsf{Ty}(\Psi.\vec{A}), D : \mathsf{Ty}(\Psi.\vec{A}.C)) \mapsto (*, \vec{A}, C : \mathsf{Ty}(\Psi.\vec{A}), D : \mathsf{Ty}(\Psi.\vec{A}.C), \Pi(C, D))$$

where $\Pi(C, D)$ is given by the Π types in $\mathbb{D}$.

Object-level Language. Finally we can define a model for the object-level language defined in the beginning of Section 6. In this object language, `trm` is indexed by `ty`. We encode the object language as follows:

$$\cdot \vdash \mathsf{ty} : \mathsf{Ty}(\top) \qquad \cdot \vdash \mathsf{trm} : \mathsf{Ty}(\top.\mathsf{ty}) \qquad \Phi : \mathsf{Ctx} \vdash \mathsf{ty}' := \mathsf{ty}\{!\} : \mathsf{Ty}(\Phi)$$

For convenience, we introduce the abbreviation ty' for $\mathsf{ty}\{!\}$. We model `ty` and `trm` in their minimal contexts in order to avoid formulating their coherence conditions w.r.t. substitutions. In the front-end language, we use the applied form `trm` a to denote a type of terms with type a. In the model given as here, we must apply a substitution instead:

$$\Phi : \mathsf{Ctx}, a : \mathtt{Tm}(\Phi, \mathsf{ty}') \vdash \mathsf{trm}\{\langle !, a\rangle\} : \mathsf{Ty}(\Phi)$$

That is, in the front-end syntax, we write `trm` a, which is interpreted as $\mathsf{trm}\{\langle !, a\rangle\}$ in the model. We introduce an abbreviation for later:

$$\mathsf{trm}[a] := \mathsf{trm}\{\langle !, a\rangle\}$$

Finally, the constructors of the object language are formulated as follows:

$$\Phi : \mathsf{Ctx} \vdash \mathsf{o} : \mathtt{Tm}(\Phi, \mathsf{ty}') \qquad \Phi : \mathsf{Ctx} \vdash \mathsf{arr} : \mathtt{Tm}(\Phi, \mathsf{ty}') \to \mathtt{Tm}(\Phi, \mathsf{ty}') \to \mathtt{Tm}(\Phi, \mathsf{ty}')$$

$$\Phi : \mathsf{Ctx} \vdash \mathsf{lam} : (a, b : \mathtt{Tm}(\Phi, \mathsf{ty}')) \to \mathtt{Tm}(\Phi.\mathsf{trm}[a], \mathsf{trm}[b\{p\}]) \to \mathtt{Tm}(\Phi, \mathsf{trm}[\mathsf{arr}(a, b)])$$

$$\Phi : \mathsf{Ctx} \vdash \mathsf{app} : (a, b : \mathtt{Tm}(\Phi, \mathsf{ty}')) \to \mathtt{Tm}(\Phi, \mathsf{trm}[\mathsf{arr}(a, b)]) \to \mathtt{Tm}(\Phi, \mathsf{trm}[a]) \to \mathtt{Tm}(\Phi, \mathsf{trm}[b])$$

Category-theoretic perspective. A number of properties of the universe `El` can be obtained by category-theoretic considerations. We have explained `El` as a syntactic representation of the Yoneda embedding. It has been shown [Capriotti 2016] that the Yoneda embedding is a morphism of CwFs, which means that it is a functor preserving the CwF structure. Using the notation of [Capriotti 2016, Definition 2.1.4] this means that there are isomorphisms such as $\mathrm{y}(\Phi.A) \cong \mathrm{y}(\Phi).\mathrm{y}^{\mathrm{Ty}}(A)$ for the preservation of context comprehension. The terms and types for the universe `El` in the internal type theory that we have defined at the beginning of this section are a syntactic presentation of this structure.

With this category-theoretic view, it is possible to use existing results on morphisms of CwFs to obtain information about the universe `El`. For example, [Clairambault and Dybjer 2014, Proposition 4.8] can be used to show that the Yoneda embedding preserves Π-types up to isomorphism, because the Yoneda embedding preserves local cartesian closed structure [Pitts 1987, Lemma 4.5]. This gives us isomorphisms such as between $\mathtt{El}\,(\Phi) \to \mathtt{El}\,(\Phi')$ and $\mathtt{El}\,(\Phi \to \Phi')$. For instance the direct proof of Lemma 6.3 above can be understood as an instance of this isomorphism.

6.4 Interpreting the Domain Level

Given the semantic model, we can detail the interpretation of the dependently typed Cocon defined in Section 6.1 into this model. The interpretation is a natural generalization of the simply typed version. First we will consider the interpretation of the domain-level types and terms, as shown in Fig. 11. One complication we encountered here is that various judgments are interdependent. For example, the type well-formedness judgment $\Gamma; \Psi \vdash A\ \mathtt{type}$ and the term well-formedness judgment $\Gamma; \Psi \vdash M : A$ depend on each other. As previously discussed, a general fact of the interpretation is that `trm` M in the syntactic level is interpreted to $\mathsf{trm}[\![M]\!]$.

In the interpretation, we proceed by interpreting domain-level types, $[\![\Gamma; \Psi \vdash A\ \mathtt{type}]\!]$. We interpret `ty` and `trm` to their semantic correspondences. Dependent function types Π is interpreted to semantic Π types as defined in the previous subsection. The interpretation of domain-level contexts $[\![\Gamma \vdash \Psi\ \mathtt{ctx}]\!]$ is defined recursively by appending interpreted types to the end of the

Interpretation of domain-level types

$$\llbracket \Gamma; \Psi \vdash \Pi x : A.B \ \texttt{type} \rrbracket = \Pi(\llbracket \Gamma; \Psi \vdash A \ \texttt{type} \rrbracket, \llbracket \Gamma; \Psi, x : A \vdash B \ \texttt{type} \rrbracket)$$

$$\llbracket \Gamma; \Psi \vdash \texttt{ty} \ \texttt{type} \rrbracket = \texttt{ty}'$$

$$\llbracket \Gamma; \Psi \vdash \texttt{trm}\ M \ \texttt{type} \rrbracket = \texttt{trm}[\llbracket \Gamma; \Psi \vdash M : \texttt{ty} \rrbracket]$$

Interpretation of domain-level contexts

$$\llbracket \Gamma \vdash \cdot \ \texttt{ctx} \rrbracket = \top$$

$$\llbracket \Gamma \vdash \Psi, x : A \ \texttt{ctx} \rrbracket = \llbracket \Gamma \vdash \Psi \ \texttt{ctx} \rrbracket . \llbracket \Gamma; \Psi \vdash A \ \texttt{type} \rrbracket$$

$$\llbracket \Gamma \vdash \psi \ \texttt{ctx} \rrbracket = \psi$$

Interpretation of domain-level substitutions where $\Psi' = \llbracket \Gamma \vdash \Psi \ \texttt{ctx} \rrbracket$

$$\llbracket \Gamma; \Psi \vdash \cdot : \cdot \rrbracket = ! : \texttt{El}\,(\Psi') \to \texttt{El}\,(\top)$$

$$\llbracket \Gamma; \Psi \vdash \sigma, M : \Phi, x : A \rrbracket = \langle e_1, e_2 \rangle : \texttt{El}\,(\Psi') \to \texttt{El}\,(\Phi'.A') \quad \text{where } A' = \llbracket \Gamma; \Phi \vdash A \ \texttt{type} \rrbracket$$
$$\text{and } \Phi' = \llbracket \Gamma \vdash \Phi \ \texttt{ctx} \rrbracket$$
$$\text{and } e_1 = \llbracket \Gamma; \Psi \vdash \sigma : \Phi \rrbracket : \texttt{El}\,(\Psi') \to \texttt{El}\,(\Phi')$$
$$\text{and } e_2 = \llbracket \Gamma; \Psi \vdash M : A[\sigma/\widehat{\Phi}] \rrbracket : \texttt{Tm}(\Psi', A'\{e_1\})$$

$$\llbracket \Gamma; \Psi, \overrightarrow{x : A} \vdash wk_{\widehat{\Psi}} : \Psi \rrbracket = p^k : \texttt{El}\,(\llbracket \Gamma \vdash \Psi, \overrightarrow{x : A} \ \texttt{ctx} \rrbracket) \to \texttt{El}\,(\Psi') \quad \text{where } k = |\overrightarrow{x{:}A}|$$

Interpretation of domain-level terms where $\Psi' = \llbracket \Gamma \vdash \Psi \ \texttt{ctx} \rrbracket$

$$\llbracket \Gamma; \Psi \vdash x : A \rrbracket = v\{p^k\} : \texttt{Tm}(\Psi', A'\{p^{k+1}\}) \quad \text{where } \Psi = \Psi_0, x : A, \overrightarrow{y_i : B_i}$$
$$\text{and } |\overrightarrow{y_i : B_i}| = k$$
$$A' = \llbracket \Gamma; \Psi_0 \vdash A \ \texttt{type} \rrbracket$$

$$\llbracket \Gamma; \Psi \vdash \lambda x.M : \Pi x : A.B \rrbracket = \Lambda(e) : \texttt{Tm}(\Psi', \Pi(A', B'))$$
$$\text{where } A' = \llbracket \Gamma; \Psi \vdash A \ \texttt{type} \rrbracket \text{ and } B' = \llbracket \Gamma; \Psi, x : A \vdash B \ \texttt{type} \rrbracket,$$
$$\text{and } e = \llbracket \Gamma; \Psi, x : A \vdash M : B \rrbracket : \texttt{Tm}(\Psi'.A', B')$$

$$\llbracket \Gamma; \Psi \vdash M\ N : [N/x]B \rrbracket = \texttt{App}(e_1, e_2) : \texttt{Tm}(\Psi', B'\{e_2\})$$
$$\text{where } A' = \llbracket \Gamma; \Psi \vdash A \ \texttt{type} \rrbracket \text{ and } B' = \llbracket \Gamma; \Psi, x : A \vdash B \ \texttt{type} \rrbracket,$$
$$\text{and } e_1 = \llbracket \Gamma; \Psi \vdash M : \Pi x : A.B \rrbracket : \texttt{Tm}(\Psi', \Pi(A', B')),$$
$$\text{and } e_2 = \llbracket \Gamma; \Psi \vdash N : A \rrbracket : \texttt{Tm}(\Psi', A)$$

$$\llbracket \Gamma; \Psi \vdash \texttt{o} : \texttt{ty} \rrbracket = \texttt{o} : \texttt{Tm}(\Psi', \texttt{ty}')$$

$$\llbracket \Gamma; \Psi \vdash \texttt{arr}\ a\ b : \texttt{ty} \rrbracket = \texttt{arr}(\llbracket \Gamma; \Psi \vdash a : \texttt{ty} \rrbracket, \llbracket \Gamma; \Psi \vdash b : \texttt{ty} \rrbracket) : \texttt{Tm}(\Psi', \texttt{ty}')$$

$$\llbracket \Gamma; \Psi \vdash \texttt{lam}\ a\ b\ m : \texttt{trm}\ (\texttt{arr}\ a\ b) \rrbracket = \texttt{lam}(a', b', \texttt{App}(e'\{p\}, v)) : \texttt{Tm}(\Psi', \texttt{trm}[arr(a', b')])$$
$$\text{where } a' = \llbracket \Gamma; \Psi \vdash a : \texttt{ty} \rrbracket : \texttt{Tm}(\Psi', \texttt{ty}'),$$
$$\text{and } b' = \llbracket \Gamma; \Psi \vdash b : \texttt{ty} \rrbracket : \texttt{Tm}(\Psi', \texttt{ty}'),$$
$$\text{and } e' = \llbracket \Gamma; \Psi \vdash m : \texttt{trm}\ a \to \texttt{trm}\ b \rrbracket : \texttt{Tm}(\Psi', \Pi(\texttt{trm}[a'], \texttt{trm}[b'\{p\}])),$$
$$\text{and } v : \texttt{Tm}(\Psi'.\texttt{trm}[a'], \texttt{trm}[a']\{p\})$$

$$\llbracket \Gamma; \Psi \vdash \texttt{app}\ a\ b\ m\ n : \texttt{trm}\ b \rrbracket = \texttt{app}(a', b', e_1, e_2) : \texttt{Tm}(\Psi', \texttt{trm}[b'])$$
$$\text{where } a' = \llbracket \Gamma; \Psi \vdash a : \texttt{ty} \rrbracket \text{ and } b' = \llbracket \Gamma; \Psi \vdash b : \texttt{ty} \rrbracket,$$
$$\text{and } e_1 = \llbracket \Gamma; \Psi \vdash m : \texttt{trm}\ (\texttt{arr}\ a\ b) \rrbracket$$
$$\text{and } e_2 = \llbracket \Gamma; \Psi \vdash n : \texttt{trm}\ a \rrbracket$$

$$\llbracket \Gamma; \Psi \vdash \lfloor t \rfloor_\sigma : [\sigma/\widehat{\Phi}]A \rrbracket = \texttt{let box}\ u = e_1\ \texttt{in}\ u\{e_2\} : \texttt{Tm}(\Psi', A'\{e_2\})$$
$$\text{where } \Phi' = \llbracket \Gamma \vdash \Phi \ \texttt{ctx} \rrbracket \text{ and } A' = \llbracket \Gamma; \Phi \vdash A \ \texttt{type} \rrbracket,$$
$$\text{and } e_1 = \llbracket \Gamma \vdash t : \lceil \Phi \vdash A \rceil \rrbracket : \flat(\texttt{Tm}(\Phi', A')),$$
$$\text{and } e_2 = \llbracket \Gamma; \Psi \vdash \sigma : \Phi \rrbracket : \texttt{El}\,(\Psi') \to \texttt{El}\,(\Phi')$$

Fig. 11. Interpretation of the domain-level level

semantic context. The empty context $\cdot$ is interpreted to the terminal object $\top$ and a context variable ψ is interpreted to an interpreted crisp variable in the interpretation of Γ, which we will give later in the interpretation of the computation level.

Domain-level substitutions are interpreted to substitution morphisms by $[\![\Gamma; \Psi \vdash \sigma : \Phi]\!]$. The interpretation in this case is very similar to the simply typed case. We use ! for the case of an empty substitution, $\langle e_1, e_2 \rangle$ for an extended substitution, and iterated first projections p^k for weakening substitutions.

Last we interpret domain-level terms using $[\![\Gamma; \Psi \vdash t : A]\!]$. In the variable case, given well-typedness, we know that the domain-level context Ψ must have the form $\Psi_0, x : A, \overrightarrow{y_i : B_i}$. We first use $v : \mathtt{Tm}(\Psi_0.A', A'\{p\})$ to extract x from $\Psi_0, x : A$. Then we apply weakening p^k to it where k is the length of the part of context after x, which gives us $v\{p^k\}$. The abstraction and the application cases are straightforward; they are interpreted to their semantic terms immediately. Next we interpret the constructors on the object level. `o` and `arr` are interpreted directly to their semantic correspondences.

The `lam` case is more interesting, because we need to determine how HOAS encoding in the domain language corresponds in the semantics. According to the rule for lam in the previous subsection, HOAS corresponds to a semantic term in the set $\mathtt{Tm}(\Phi'.\mathtt{trm}[a'], \mathtt{trm}[b'\{p\}])$. Meanwhile, if we directly interpret m as in the rule, we obtain a term e' in the set $\mathtt{Tm}(\Psi', \Pi(\mathtt{trm}[a'], \mathtt{trm}[b'\{p\}]))$. Therefore, we need to transform e' properly by supplying $\mathsf{App}(e'\{p\}, v)$. We can examine that this transformation does achieve the goal:

$$\begin{aligned}\mathsf{App}(e'\{p\}, v) &: \mathtt{Tm}(\Psi'.\mathtt{trm}[a'], \mathtt{trm}[b'\{p\}]\{p\}\{\langle id_{\Psi'.\mathtt{trm}[a']}, v\rangle\}) \\ &= \mathtt{Tm}(\Psi'.\mathtt{trm}[a'], \mathtt{trm}[b'\{p\}]\{p \circ \langle id_{\Psi'.\mathtt{trm}[a']}, v\rangle\}) \\ &= \mathtt{Tm}(\Psi'.\mathtt{trm}[a'], \mathtt{trm}[b'\{p\}])\end{aligned}$$

The `app` case is straightforward. In the unbox case, we first interpret t, from which we obtain a boxed semantic term. We use `let box` to extract from it u in $\mathtt{Tm}(\Phi', A')$. By applying the interpreted substitution e_2 to u, we obtain a term in the expected set $\mathtt{Tm}(\Psi', A'\{e_2\})$.

6.5 Interpreting the Computation Level

In this section, we discuss the interpretation of the computation level of Cocon. The interpretation of the computation level is simpler than the one of the domain level, because we work in a presheaf category which possesses a CwF structure with Π types. The interpretation only needs to relate the corresponding parts.

The interpretation functions without the recursors are shown in Fig. 12. They are very similar to the simply typed case. In the interpretation of computation types, $[\![\Gamma \vdash \check{\tau}\ \ \mathtt{type}]\!]$, we surround boxed contextual types with a $\flat$ modality, and the contextual types are interpreted using $[\![\Gamma \vdash (\Psi \vdash A)]\!]$, resulting in some Tm. Computation-level functions are directly translated to crisp functions in the model, similar to the simply typed case. This is because we want to maintain the invariant where computation-level variables are all crisp and in the model all crisp variables live in $\flat$. At last, we simply map `ctx` to Ctx which is a type representing the domain-level contexts.

The interpretation of computation contexts, $[\![\Gamma]\!]$, simply iteratively interprets all types in it. Note that a computation-level context Γ is interpreted to a global or crisp context in our model. That is why we do not wrap all interpreted types with $\flat$ and seemingly mismatches with the parameter types in function types. We will resolve this problem when interpreting computation-level terms.

The interpretations of contextual objects and contextual terms are immediately reduced to the interpretations of domain-level types and terms, which we have discussed in the previous subsection. Since $\Psi \vdash_v A$ denotes a variable of type A in domain context Ψ, the interpretation as a

$$
\begin{array}{rl}
\text{Intepretation of computation types} & \\
[\![\Gamma \vdash \lceil T \rceil \ \mathtt{type}]\!] & = \flat [\![\Gamma \vdash T]\!] \\
[\![\Gamma \vdash (x : \check{\tau}_1) \Rightarrow \tau_2 \ \mathtt{type}]\!] & = (x :: [\![\Gamma \vdash \check{\tau}_1 \ \mathtt{type}]\!]) \to^{\flat} [\![\Gamma, x : \check{\tau}_1 \vdash \tau_2 \ \mathtt{type}]\!] \\
[\![\Gamma \vdash \mathtt{ctx} \ \mathtt{type}]\!] & = \mathsf{Ctx} \\
\text{Interpretation of computation contexts} & \\
[\![\cdot]\!] & = \cdot \\
[\![\Gamma, x : \check{\tau}]\!] & = [\![\Gamma]\!], x :: [\![\Gamma \vdash \check{\tau} \ \mathtt{type}]\!] \\
\text{Interpretation of contextual objects} & \\
[\![\Gamma \vdash (\widehat{\Psi} \vdash M) : (\Psi \vdash A)]\!] & = [\![\Gamma; \Psi \vdash M : A]\!] \\
[\![\Gamma \vdash (\widehat{\Psi} \vdash M) : (\Psi \vdash_v A)]\!] & = [\![\Gamma; \Psi \vdash M : A]\!] \\
\text{Interpretation of contextual types} & \\
[\![\Gamma \vdash (\Psi \vdash A)]\!] & = \mathsf{Tm}([\![\Gamma \vdash \Psi \ \mathtt{ctx}]\!], [\![\Gamma; \Psi \vdash A \ \mathtt{type}]\!]) \\
[\![\Gamma \vdash (\Psi \vdash_v A)]\!] & = \mathsf{Tm}_v([\![\Gamma \vdash \Psi \ \mathtt{ctx}]\!], [\![\Gamma; \Psi \vdash A \ \mathtt{type}]\!]) \quad \mathsf{Tm}_v \text{ is } \mathsf{Tm} \text{ but equivalent to } v\{p^k\} \\
\text{Interpretation of computation terms} & \\
[\![\Gamma \vdash \lceil C \rceil : \lceil T \rceil]\!] & = \mathsf{box}\, [\![\Gamma \vdash C : T]\!] \\
[\![\Gamma \vdash t\ s : [s/y]\tau_2]\!] & = [\![\Gamma \vdash t : (y : \check{\tau}_1) \Rightarrow \tau_2]\!]\, [\![\Gamma \vdash s : \check{\tau}_1]\!] \\
[\![\Gamma \vdash \mathtt{fn}\ x \Rightarrow t : (x : \check{\tau}_1) \Rightarrow \tau_2]\!] & = \between^{\flat} x :: \flat [\![\Gamma \vdash \check{\tau}_1 \ \mathtt{type}]\!].[\![\Gamma, x : \check{\tau}_1 \vdash t : \tau_2]\!] \\
[\![\Gamma \vdash x : \check{\tau}]\!] & = x
\end{array}
$$

Fig. 12. Interpretation of the computation level without recursors

$\mathtt{Tm}(\Phi, A)$ is restricted such that it is semantically also a variable lookup $v\{p^k\}$. restricted semantic term in the set $\mathtt{Tm}(\Phi, A)$ in the form of $v\{p^k\}$. This is indeed the case by looking at the interpretation of the variable case of the domain level.

The interpretation of the computation-level terms is straightforward. Boxed contextual objects are simply interpreted as boxed domain-level terms in the model. Since we interpret to computation-level functions to crisp functions, we use crisp applications and $\between^{\flat}$ abstractions respectively to interpret computation-level applications and abstractions. Notice that in the application case, due to the soundness theorem we are about to show, the interpretation of s, $[\![\Gamma \vdash s : \check{\tau}_1]\!]$, is indeed closed, and thus the application is valid. At last, variables are simply interpreted to those in the semantic context.

6.6 Interpreting Recursors

Compared to the previous standard interpretations, the recursors are more interesting to consider. The recursor of `ty` appears to be very typical because it is simply a algebraic data type, which can already be modeled conventionally using the initial algebra of some polynomial functor. Therefore, we omit the concrete formulation here in favor of conciseness and only focus on the recursor of `trm`.

The semantic recursor for trm:

$$\frac{\Gamma \mid \cdot \vdash \Psi : \mathsf{Ctx} \qquad \Gamma \mid \cdot \vdash a : \flat(\mathsf{Tm}(\top, \mathsf{ty})) \qquad \Gamma \mid \cdot \vdash y : \flat(\mathsf{Tm}(\Psi, \mathsf{trm}[\mathtt{lift}(\Psi, a)]))}{\Gamma \vdash R(\Psi, a, y)\ \ \mathsf{type}}$$

$$\frac{\Gamma \mid \cdot \vdash \Psi : \mathsf{Ctx} \qquad \Gamma \mid \cdot \vdash a : \flat(\mathsf{Tm}(\top, \mathsf{ty})) \qquad \Gamma \mid \cdot \vdash t : \flat(\mathsf{Tm}_{\mathcal{V}}(\Psi, \mathsf{trm}[\mathtt{lift}(\Psi, a)]))}{\Gamma \mid \cdot \vdash B_{\mathcal{V}}(\Psi, a, t) : R(\Psi, a, t)}$$

$$\frac{\begin{array}{c}\Gamma \mid \cdot \vdash \Psi : \mathsf{Ctx} \qquad \Gamma \mid \cdot \vdash a : \flat(\mathsf{Tm}(\top, \mathsf{ty})) \\ \Gamma \mid \cdot \vdash b : \flat(\mathsf{Tm}(\top, \mathsf{ty})) \qquad \Gamma \mid \cdot \vdash m : \flat(\mathsf{Tm}(\Psi.\mathsf{trm}[\mathtt{lift}(\Psi, a)], \mathsf{trm}[\mathtt{lift}(\Psi.\mathsf{trm}[\mathtt{lift}(\Psi, a)], b)])) \\ \Gamma \mid \cdot \vdash f_m : R(\Psi.\mathsf{trm}[\mathtt{lift}(\Psi, a)], b, m)\end{array}}{\Gamma \mid \cdot \vdash B_{\mathsf{lam}}(\Psi, m, f_m) : R(\Phi, \mathsf{arr}'(a, b), \mathsf{lam}'(\Psi, m))}$$

$$\frac{\begin{array}{c}\Gamma \mid \cdot \vdash \Psi : \mathsf{Ctx} \qquad \Gamma \mid \cdot \vdash a : \flat(\mathsf{Tm}(\top, \mathsf{ty})) \qquad \Gamma \mid \cdot \vdash b : \flat(\mathsf{Tm}(\top, \mathsf{ty})) \\ \Gamma \mid \cdot \vdash m : \flat(\mathsf{Tm}(\Psi, \mathsf{trm}[\mathsf{arr}(\mathtt{lift}(\Psi, a), \mathtt{lift}(\Psi, b))])) \\ \Gamma \mid \cdot \vdash n : \flat(\mathsf{Tm}(\Psi, \mathsf{trm}[\mathtt{lift}(\Psi, a)])) \qquad \Gamma \mid \cdot \vdash f_m : R(\Psi, \mathsf{arr}'(a, b), m) \qquad \Gamma \mid \cdot \vdash f_n : R(\Psi, a, n)\end{array}}{\Gamma \mid \cdot \vdash B_{\mathsf{app}}(\Psi, m, n, f_m, f_n) : R(\Psi, b, \mathsf{app}'(\Psi, m, n))}$$

$$\Gamma \mid \cdot \vdash rec_{\mathsf{trm}}(B_{\mathcal{V}}, B_{\mathsf{lam}}, B_{\mathsf{app}}) : (\Psi : \mathsf{Ctx}) \to (a : \flat(\mathsf{Tm}(\top, \mathsf{ty}))) \to (y : \flat(\mathsf{Tm}(\Psi, \mathsf{trm}[\mathtt{lift}(\Psi, a)]))) \to R(\Psi, a, y)$$

Equations:

$$\begin{aligned}
&rec_{\mathsf{trm}}(B_{\mathcal{V}}, B_{\mathsf{lam}}, B_{\mathsf{app}}, \Psi, a, x) = B_{\mathcal{V}}(\Psi, a, x) && \text{where } x : \flat(\mathsf{Tm}_{\mathcal{V}}(\Psi, \mathsf{trm}[\mathtt{lift}(\Psi, a)])) \\
&rec_{\mathsf{trm}}(B_{\mathcal{V}}, B_{\mathsf{lam}}, B_{\mathsf{app}}, \Psi, \mathsf{arr}'(a, b), \mathsf{lam}'(\Psi, m)) \\
=&B_{\mathsf{lam}}(\Psi, f, rec_{\mathsf{trm}}(B_{\mathcal{V}}, B_{\mathsf{lam}}, B_{\mathsf{app}}, \Psi.\mathsf{trm}[\mathtt{lift}(\Psi, a)], b, m)) \\
&rec_{\mathsf{trm}}(B_{\mathcal{V}}, B_{\mathsf{lam}}, B_{\mathsf{app}}, \Psi, b, \mathsf{app}'(\Psi, m, n)) \\
=&B_{\mathsf{lam}}(\Psi, f, m, n, rec_{\mathsf{trm}}(B_{\mathcal{V}}, B_{\mathsf{lam}}, B_{\mathsf{app}}, \Psi, \mathsf{arr}'(a, b), m), rec_{\mathsf{trm}}(B_{\mathcal{V}}, B_{\mathsf{lam}}, B_{\mathsf{app}}, \Psi, a, n))
\end{aligned}$$

Fig. 13. Semantic recursor

In order to formulate the semantic recursor of trm, we define the following three auxiliary definitions:

$$\frac{\Gamma \mid \cdot \vdash A : \mathsf{Ty}(\top) \qquad \Gamma \mid \cdot \vdash \Psi : \mathsf{Ctx} \qquad \Gamma \mid \cdot \vdash t : \flat(\mathsf{Tm}(\top, A))}{\Gamma \mid \cdot \vdash \mathtt{lift}(\Psi, t) := \mathtt{let\ box}\ x' = t\ \mathtt{in}\ x'\{!\} : \mathsf{Tm}(\Psi, A\{!\})}$$

$$\frac{\Gamma \mid \cdot \vdash a : \flat(\mathsf{Tm}(\top, \mathsf{ty})) \qquad \Gamma \mid \cdot \vdash b : \flat(\mathsf{Tm}(\top, \mathsf{ty}))}{\Gamma \mid \cdot \vdash \mathsf{arr}'(a, b) := \mathtt{let\ box}\ a' = a\ \mathtt{in\ let\ box}\ b' = b\ \mathtt{in\ box}\ \mathsf{arr}(a', b') : \flat(\mathsf{Tm}(\top, \mathsf{ty}))}$$

$$\frac{\begin{array}{c}\Gamma \mid \cdot \vdash \Psi : \mathsf{Ctx} \\ \Gamma \mid \cdot \vdash a : \mathsf{Tm}(\Psi, \mathsf{ty}') \qquad \Gamma \mid \cdot \vdash b : \mathsf{Tm}(\Psi, \mathsf{ty}') \qquad \Gamma \mid \cdot \vdash m : \flat(\mathsf{Tm}(\Psi.\mathsf{trm}[a], \mathsf{trm}[b\{p\}]))\end{array}}{\Gamma \mid \cdot \vdash \mathsf{lam}'(\Psi, m) := \mathtt{let\ box}\ m' = m\ \mathtt{in\ box}\ \mathsf{lam}(a, b, m') : \flat(\mathsf{Tm}(\Psi, \mathsf{trm}[\mathsf{arr}(a, b)]))}$$

$$\frac{\begin{array}{c}\Gamma \mid \cdot \vdash \Psi : \mathsf{Ctx} \qquad \Gamma \mid \cdot \vdash a : \mathsf{Tm}(\Psi, \mathsf{ty}') \\ \Gamma \mid \cdot \vdash b : \mathsf{Tm}(\Psi, \mathsf{ty}') \qquad \Gamma \mid \cdot \vdash m : \flat(\mathsf{Tm}(\Psi, \mathsf{trm}[arr(a, b)])) \qquad \Gamma \mid \cdot \vdash n : \flat(\mathsf{Tm}(\Psi, \mathsf{trm}[a]))\end{array}}{\Gamma \mid \cdot \vdash \mathsf{app}'(\Psi, m, n) := \mathtt{let\ box}\ m' = m\ \mathtt{in\ let\ box}\ n' = n\ \mathtt{in\ box}\ \mathsf{app}(a, b, m', n') : \flat(\mathsf{Tm}(\Psi, \mathsf{trm}[b]))}$$

These helpers are defined to ease the operations related to the $\flat$ modality. For example, the `lift` function transforms a term t of type $\flat(\mathsf{Tm}(\top, A))$ to $\mathsf{Tm}(\Psi, A\{!\})$ for some domain-level context Ψ.

The interpretation of the recursor for trm where $\mathcal{I} = (\psi : \mathtt{ctx}) \Rightarrow (z : \lceil \vdash \mathtt{ty} \rceil) \Rightarrow (y : \lceil \psi \vdash \mathtt{trm}\ \lfloor z \rfloor . \rceil) \Rightarrow \tau$

$$[\![\Gamma \vdash \mathtt{rec}^{\mathcal{I}}\ (b_v \mid b_{\mathtt{lam}} \mid b_{\mathtt{app}})\ \Psi\ t\ t' : \tau[\Psi, t, t'/\psi, z, y]]\!] = rec_{\mathtt{trm}}(e_v, e_{\mathtt{lam}}, e_{\mathtt{app}}, e_\Psi, e_t, e_{t'})$$

where $e_\Psi = [\![\Gamma \vdash \Psi\ \mathtt{ctx}]\!]$

and $e_t = [\![\Gamma \vdash t : \lceil \vdash \mathtt{ty} \rceil]\!]$

and $e_{t'} = [\![\Gamma \vdash t' : \lceil \Psi \vdash \mathtt{trm}[\lfloor t \rfloor .]\rceil]\!]$

and $e_v = [\![\Gamma \vdash b_v : \mathcal{I}]\!]$

and $e_{\mathtt{lam}} = [\![\Gamma \vdash b_{\mathtt{lam}} : \mathcal{I}]\!]$

and $e_{\mathtt{app}} = [\![\Gamma \vdash b_{\mathtt{app}} : \mathcal{I}]\!]$

Interpretation of branches

$$[\![\Gamma \vdash b_v : \mathcal{I}]\!] = \curlywedge^{\flat}\ \Psi\ a\ t.e \quad \text{where } e = [\![\Gamma, \Psi : \mathtt{ctx}, a : \lceil \vdash \mathtt{ty} \rceil, t : \lceil \Psi \vdash_v \mathtt{trm}\ \lfloor a \rfloor . \rceil \vdash t_v : \tau[a, t/z, y]]\!]$$

$$[\![\Gamma \vdash b_{\mathtt{lam}} : \mathcal{I}]\!] = \curlywedge^{\flat}\ \Psi\ a\ b\ m\ f_m.e \quad \text{where } e = [\![\text{the premise judgment}]\!]$$

$$[\![\Gamma \vdash b_{\mathtt{app}} : \mathcal{I}]\!] = \curlywedge^{\flat}\ \Psi\ a\ b\ m\ n\ f_m\ f_n.e \quad \text{where } e = [\![\text{the premise judgment}]\!]$$

Fig. 14. Interpretation of recursor for trm

lam′ takes a boxed HOAS representation, m, and return a boxed trm constructed by lam. Similarly, app′ takes two boxed trm and return a boxed trm constructed by app. We need these helpers in order to reduce the clusters in the formulation of the semantic recursor of `trm` and the equations, which is presented in Fig. 13.

Since the recursion happens in the computation level, we require the local context to be empty, so we only handle closed domain-level types and terms. In the variable case, we require the semantic term t to be Tm_v, so that it indeed represents a variable in Ψ. In the lam case, the recursion involves a HOAS encoding of an object-level term. This corresponds to a domain-level term with an augmented context $\Psi.\mathtt{trm}[\mathtt{lift}(\Psi, a)]$. The app case is straightforward since it just goes down to the subterms recursively.

Given the semantic recursor, we can straightforwardly interpret the syntactical recursor, shown in Fig. 14.

Given the recursor, we can interpret the syntactical recursor quite straightforwardly to the semantic recursor. Following the pattern from the simply typed case, we interpret branches to crisp functions and the bodies are recursively interpreted.

That concludes our interpretations. We formulate the soundness properties of the interpretations as below, which are proved via a mutual induction.

Theorem 6.4 (Soundness). *The following are true.*

(1) *If* $\Gamma \vdash \Phi\ \mathtt{ctx}$, *then* $[\![\Gamma]\!] \mid \cdot \vdash [\![\Gamma \vdash \Phi\ \mathtt{ctx}]\!] : \mathsf{Ctx}$.
(2) *If* $\Gamma; \Psi \vdash A\ \mathtt{type}$, *then* $[\![\Gamma]\!] \mid \cdot \vdash [\![\Gamma; \Psi \vdash A\ \mathtt{type}]\!] : \mathsf{Ty}([\![\Gamma \vdash \Psi\ \mathtt{ctx}]\!])$.
(3) *If* $\Gamma; \Psi \vdash M : A$, *then* $[\![\Gamma]\!] \mid \cdot \vdash [\![\Gamma; \Psi \vdash M : A]\!] : \mathsf{Tm}([\![\Gamma \vdash \Psi\ \mathtt{ctx}]\!], [\![\Gamma; \Psi \vdash A\ \mathtt{type}]\!])$.
(4) *If* $\Gamma; \Psi \vdash \sigma : \Phi$, *then* $[\![\Gamma]\!] \mid \cdot \vdash [\![\Gamma; \Psi \vdash \sigma : \Phi]\!] : \mathsf{El}\,([\![\Gamma \vdash \Psi\ \mathtt{ctx}]\!]) \to \mathsf{El}\,([\![\Gamma \vdash \Phi\ \mathtt{ctx}]\!])$.
(5) *If* $\Gamma \vdash C : T$, *then* $[\![\Gamma]\!] \mid \cdot \vdash [\![\Gamma \vdash C : T]\!] : [\![\Gamma \vdash T]\!]$.
(6) *If* $\Gamma \vdash \check{\tau}\ \mathtt{type}$, *then* $[\![\Gamma]\!] \mid \cdot \vdash [\![\Gamma \vdash \check{\tau}\ \mathtt{type}]\!]\ \mathtt{type}$.
(7) *If* $\Gamma \vdash t : \check{\tau}$, *then* $[\![\Gamma]\!] \mid \cdot \vdash [\![\Gamma \vdash t : \check{\tau}]\!] : [\![\Gamma \vdash \check{\tau}\ \mathtt{type}]\!]$.
(8) *If* $\Gamma; \Psi \vdash A \equiv A'\ \mathtt{type}$, *then* $[\![\Gamma]\!] \mid \cdot \vdash [\![\Gamma; \Psi \vdash A\ \mathtt{type}]\!] = [\![\Gamma; \Psi \vdash A'\ \mathtt{type}]\!] : \mathsf{Ty}([\![\Gamma \vdash \Psi\ \mathtt{ctx}]\!])$.

(9) *If* $\Gamma;\ \Psi \vdash M \equiv N : A$ *then* $[\![\Gamma]\!] \mid \cdot \vdash [\![\Gamma;\ \Psi \vdash M : A]\!] = [\![\Gamma;\ \Psi \vdash N : A]\!] : \mathtt{Tm}([\![\Gamma \vdash \Psi\ \mathtt{ctx}]\!], [\![\Gamma; \Psi \vdash A\ \mathtt{type}]\!])$.
(10) *If* $\Gamma; \Psi \vdash \sigma \equiv \sigma' : \Phi$ *then* $[\![\Gamma]\!] \mid \cdot \vdash [\![\Gamma; \Psi \vdash \sigma : \Phi]\!] = [\![\Gamma; \Psi \vdash \sigma' : \Phi]\!] : \mathtt{El}\,([\![\Gamma \vdash \Psi\ \mathtt{ctx}]\!]) \to \mathtt{El}\,([\![\Gamma \vdash \Phi\ \mathtt{ctx}]\!])$
(11) *If* $\Gamma \vdash \check{\tau}_1 \equiv \check{\tau}_2\ \mathtt{type}$, *then* $[\![\Gamma]\!] \mid \cdot \vdash [\![\Gamma \vdash \check{\tau}_1\ \mathtt{type}]\!] = [\![\Gamma \vdash \check{\tau}_2\ \mathtt{type}]\!]\ \mathtt{type}$.
(12) *If* $\Gamma \vdash t_1 \equiv t_2 : \tau$, $[\![\Gamma]\!] \mid \cdot \vdash [\![\Gamma \vdash t_1 : \check{\tau}]\!] = [\![\Gamma \vdash t_2 : \check{\tau}]\!] : [\![\Gamma \vdash \check{\tau}\ \mathtt{type}]\!]$.

7 CONNECTION WITH FITCH-STYLE TYPE THEORIES

Birkedal et al. [2020]; Gratzer et al. [2019] discussed two Fitch-style dependent modal type theories. Compared to the dual-context-style we presented in previous sections, Fitch-style systems differ in that they use only one context to keep track of all variables. Instead, Fitch-style systems use a "locking" mechanism to prevent variable lookups from continuing. In this section, we establish a relation between a fragment of Cocon without recursion on HOAS and a semantic framework discussed in Birkedal et al. [2020], dependent right adjoints. We show this by showing an embedding of Cocon into the dependent intuitionistic K shown in Birkedal et al. [2020], which has the soundness and completeness properties with respect to dependent right adjoints. Thus we can establish that the fragment of Cocon without recursion on HOAS can be interpreted by any system with dependent right adjoints.

7.1 Fitch-style Modal Type Theories

Fitch-style modal type theories are more intuitive than dual-context-style modal type theories in a sense that Fitch style only handles one context. As a consequence, valid and true assumptions in the contexts are mixed together. The introduction and the elimination rules thus must tell these different kinds of assumption apart. In Fitch-style systems, the introduction rule for necessity or box introduces a lock to the top of the context. This lock "blocks" the context to its left. Different flavors of Fitch-style systems are distinguished by their elimination rules. For dependent intuitionistic K, the rules are

$$\frac{\Gamma, 🔒 \vdash m : T}{\Gamma \vdash \mathsf{box}\ m : \Box T} \qquad \frac{\Gamma \vdash m : \Box T \qquad 🔒 \notin \Gamma'}{\Gamma, 🔒, \Gamma' \vdash \mathsf{unbox}\ m : T}$$

In both rules, 🔒 symbol prevents variable lookups from going beyond its left. That is, a term can only refer to variables to the right of the rightmost 🔒. The rules can be understood from the classical Kripke's semantics: box accesses the next world, and thus the 🔒 symbol locks the assumptions in all previous worlds (to its left); while unbox allows us to only travel back to the immediate previous world, so 🔒 must not exist in Γ' in the elimination rule. Cocon is much closer to dependent intuitionistic K than to the dual-context models we discussed in the previous sections. Nonetheless, one fundamental problem of dependent intuitionistic K is that it does not support recursion on HOAS structures like `trm` in Section 6 as Cocon does. Therefore, in this section, we will only discuss the interpretation from the fragment of Cocon without recursors to dependent intuitionistic K in order to discuss their connection explicitly. Whether dependent intuitionistic K can be extended to support recursion on HOAS is an interesting topic for future investigation.

7.2 Dependent Right Adjoints

Dependent right adjoints are a special structure added on top of a category with families. Essentially it is a CwF equipped with a functor L, denoting the 🔒 symbol, and a family of types R, denoting the modality. Dependent right adjoints are used to capture the nature of comonadic modality in a categorical language.

Definition 7.1. A category with families with a dependent right adjoint C is a category with families with the following extra data:

(1) an endofunctor $L : \mathcal{C} \to \mathcal{C}$,
(2) a family $R_\Gamma(A) \in Ty(\Gamma)$ for each $\Gamma \in \mathcal{C}$ and $A \in Ty(L(\Gamma))$.

The following axioms hold:

(1) $R_\Gamma(A)\{\sigma\} = R_\Delta(A\{L(\sigma)\}) \in Ty(\Delta)$ for $\sigma : \Delta \to \Gamma$,
(2) the following isomorphism exists for $\Gamma \in \mathcal{C}$ and $A \in Ty(L(\Gamma))$:

$$Tm(L(\Gamma), A) \simeq Tm(\Gamma, R_\Gamma(A))$$

with the effect from left to right as $\overrightarrow{M} \in Tm(\Gamma, R_\Gamma(A))$ for $M \in Tm(L(\Gamma), A)$ and the other effect $\overleftarrow{N} \in Tm(L(\Gamma), A)$ for $N \in Tm(\Gamma, R_\Gamma(A))$.
(3) $\overrightarrow{M}\{\sigma\} = \overrightarrow{M\{L(\sigma)\}} \in Tm(\Delta, R_\Delta(A\{\sigma\}))$ for $M \in Tm(L(\Gamma), A)$ and $\sigma \in \Delta \to \Gamma$.

Birkedal et al. [2020] shows that dependent intuitionistic K can be soundly interpreted into a CwF with a dependent right adjoint and the type theory itself forms a term model. Thus it suffices to interpret Cocon to their type theory to show that Cocon can be interpreted using the structure of a dependent right adjoint. We give the interpretation in the next section.

7.3 Interpreting Cocon

The interpretation is mostly straightforward. We can easily show that the interpretation is also sound:

Theorem 7.2 (Soundness). *The following are true.*

(1) *If* $\Gamma \vdash \Phi\ \mathtt{ctx}$, *then* $[\![\Gamma]\!] \vdash [\![\Gamma \vdash \Phi\ \mathtt{ctx}]\!] : \mathsf{Ctx}$.
(2) *If* $\Gamma; \Psi \vdash A\ \mathtt{type}$, *then* $[\![\Gamma]\!] \vdash [\![\Gamma; \Psi \vdash A\ \mathtt{type}]\!] : \Pi u : [\![\Gamma \vdash \Psi\ \mathtt{ctx}]\!].\mathsf{Type}$.
(3) *If* $\Gamma; \Psi \vdash M : A$, *then* $[\![\Gamma]\!] \vdash [\![\Gamma; \Psi \vdash M : A]\!] : \Pi u : [\![\Gamma \vdash \Psi\ \mathtt{ctx}]\!].[\![\Gamma; \Psi \vdash A\ \mathtt{type}]\!]\ u$.
(4) *If* $\Gamma; \Psi \vdash \sigma : \Phi$, *then* $[\![\Gamma]\!] \vdash [\![\Gamma; \Psi \vdash \sigma : \Phi]\!] : \Pi u : [\![\Gamma \vdash \Psi\ \mathtt{ctx}]\!].[\![\Gamma \vdash \Phi\ \mathtt{ctx}]\!]$.
(5) *If* $\Gamma \vdash C : T$, *then* $[\![\Gamma]\!] \vdash [\![\Gamma \vdash C : T]\!] : [\![\Gamma \vdash T]\!]$.
(6) *If* $\Gamma \vdash \check{\tau}\ \mathtt{type}$, *then* $[\![\Gamma]\!] \vdash [\![\Gamma \vdash \check{\tau}\ \mathtt{type}]\!] : \mathsf{Type}$.
(7) *If* $\Gamma \vdash t : \check{\tau}$, *then* $[\![\Gamma]\!] \vdash [\![\Gamma \vdash t : \check{\tau}]\!] : [\![\Gamma \vdash \check{\tau}\ \mathtt{type}]\!]$.

There are a number of differences to highlight:

(1) In Cocon, the domain level sees contextual variables on the computation level. This implies that contextual variables must live in $\Box$.
(2) The domain level and the computation level in Cocon have different syntax. In the interpretation, we need to merge them, e.g. two dependent function spaces are merged into the same syntax. We can still distinguish them by looking at the level they live in.

This idea leads to an interpretation shown in Fig. 15. As we can see, the interpretation is very straightforward, showing that the Fitch-style system is compatible with Cocon in many aspects.

In the interpretation of the domain-level types, we only show the case for Π types. If we have corresponding base types in Cocon and dependent intuitionistic K, we can relate them via the interpretation. For example, if we have ty in dependent intuitionistic K as well, then we will have a base case in the interpretation. Nonetheless, we can still work on other parts of the interpretations.

In dependent intuitionistic K, we assume a universe Ctx, which is used to represent domain-level contexts. There are two types, $\top$ to represent the empty context and $-.-$ for an extended context. That is, a domain-level context are managed as a list-like structure. To construct a domain-level context in dependent intuitionistic K, we use () to construct an empty context and $-,-$ to extend an existing context. Given $\Phi.A$, we can get the precedent Φ by applying π_1 and get the domain-level term of type A by applying π_2. This is sufficient for us to perform operations related domain-level contexts in dependent intuitionistic K. The interpretation of domain-level contexts

Interpretation of domain-level types

$[\![\Gamma; \Psi \vdash \Pi x : A.B \ \texttt{type}]\!] = \curlywedge u.\Pi x : [\![\Gamma; \Psi \vdash A \ \texttt{type}]\!]\ u.[\![\Gamma; \Psi, x : A \vdash B \ \texttt{type}]\!]\ u$

Interpretation of domain-level contexts

$[\![\Gamma \vdash \cdot \ \texttt{ctx}]\!] = \top$

$[\![\Gamma \vdash \Psi, x : A \ \texttt{ctx}]\!] = [\![\Gamma \vdash \Psi \ \texttt{ctx}]\!].[\![\Gamma; \Psi \vdash A \ \texttt{type}]\!]$

$[\![\Gamma \vdash \psi \ \texttt{ctx}]\!] = \mathsf{unbox}\ \psi$

Interpretation of domain-level substitutions where $\Psi' = [\![\Gamma \vdash \Psi \ \texttt{ctx}]\!]$

$[\![\Gamma; \Psi \vdash \cdot : \cdot]\!] = \curlywedge u.()$

$[\![\Gamma; \Psi \vdash \sigma, M : \Phi, x : A]\!] = \curlywedge u.(e_1\ u, e_2\ u)$ where $e_1 = [\![\Gamma; \Psi \vdash \sigma : \Phi]\!]$ and $e_2 = [\![\Gamma; \Psi \vdash M : A[\sigma/\widehat{\Phi}]]\!]$

$[\![\Gamma; \Psi, \overrightarrow{x : A} \vdash wk_{\widehat{\Psi}} : \Psi]\!] = \pi_1^k$ where $k = |\overrightarrow{x{:}A}|$

Interpretation of domain-level terms where $\Psi' = [\![\Gamma \vdash \Psi \ \texttt{ctx}]\!]$

$[\![\Gamma; \Psi \vdash x : A]\!] = \curlywedge u.\pi_2(\pi_1^k\ u)$ where $\Psi = \Psi_0, x : A, \overrightarrow{y_i : B_i}$ and $|\overrightarrow{y_i : B_i}| = k$

$[\![\Gamma; \Psi \vdash \lambda x.M : \Pi x : A.B]\!] = \curlywedge u\ x.e\ (u, x)$ where $e = [\![\Gamma; \Psi, x : A \vdash M : B]\!]$

$[\![\Gamma; \Psi \vdash M\ N : [N/x]B]\!] = \curlywedge u.(e_1\ u)\ (e_2\ u)$ where $e_1 = [\![\Gamma; \Psi \vdash M : \Pi x : A.B]\!]$ and $e_2 = [\![\Gamma; \Psi \vdash N : A]\!]$

$[\![\Gamma; \Psi \vdash \lfloor t \rfloor_\sigma : [\sigma/\widehat{\Phi}]A]\!] = \curlywedge u.(\mathsf{unbox}\ e_1)\ (e_2\ u)$ where $e_1 = [\![\Gamma \vdash t : \lceil \Phi \vdash A \rceil]\!]$ and $e_2 = [\![\Gamma; \Psi \vdash \sigma : \Phi]\!]$

Interpretation of contextual objects

$[\![\Gamma \vdash (\widehat{\Psi} \vdash M) : (\Psi \vdash A)]\!] = [\![\Gamma; \Psi \vdash M : A]\!]$

$[\![\Gamma \vdash (\widehat{\Psi} \vdash M) : (\Psi \vdash_v A)]\!] = [\![\Gamma; \Psi \vdash M : A]\!]$

Interpretation of contextual types

$[\![\Gamma \vdash (\Psi \vdash A)]\!] = \Pi u : [\![\Gamma \vdash \Psi \ \texttt{ctx}]\!].[\![\Gamma; \Psi \vdash A \ \texttt{type}]\!]$

$[\![\Gamma \vdash (\Psi \vdash_v A)]\!] = \Pi u : [\![\Gamma \vdash \Psi \ \texttt{ctx}]\!].[\![\Gamma; \Psi \vdash A \ \texttt{type}]\!]$

Intepretation of computation types

$[\![\Gamma \vdash \lceil T \rceil \ \texttt{type}]\!] = \Box[\![\Gamma \vdash T]\!]$

$[\![\Gamma \vdash (x : \check{\tau}_1) \Rightarrow \tau_2 \ \texttt{type}]\!] = \Pi x : [\![\Gamma \vdash \check{\tau}_1 \ \texttt{type}]\!].[\![\Gamma \vdash \tau_2 \ \texttt{type}]\!]$

$[\![\Gamma \vdash \texttt{ctx type}]\!] = \Box\mathsf{Ctx}$

Interpretation of computation contexts

$[\![\cdot]\!] = \cdot$

$[\![\Gamma, x : \check{\tau}]\!] = [\![\Gamma]\!], x : [\![\Gamma \vdash \check{\tau} \ \texttt{type}]\!]$

Interpretation of computation terms

$[\![\Gamma \vdash \lceil C \rceil : \lceil T \rceil]\!] = \mathsf{box}\ [\![\Gamma \vdash C : T]\!]$

$[\![\Gamma \vdash t_1\ t_2 : [t_2/x]\tau]\!] = [\![\Gamma \vdash t_1 : (x : \check{\tau}_2) \Rightarrow \tau]\!]\ [\![\Gamma \vdash t_2 : \check{\tau}_2]\!]$

$[\![\Gamma \vdash \mathsf{fn}\ x \Rightarrow t : (x : \check{\tau}_1) \Rightarrow \tau_2]\!] = \curlywedge x : [\![\check{\tau}_1]\!].[\![\Gamma, x : \check{\tau}_1 \vdash t : \tau_2]\!]$

$[\![\Gamma \vdash x : \check{\tau}]\!] = x$

Fig. 15. Interpretation to the Fitch-style system

is quite straightforward, except that when we encounter a context variable ψ, we unbox it in the interpretation. This is because contextual variables always have type $\Box\mathsf{Ctx}$ in the model, as to be shown in the interpretation of computation-level types.

The interpretations of domain-level substitutions and terms are also very straightforward. Notice that the unbox case in the interpretation of terms becomes much more direct. Since the interpretation of t is some boxed function type, we simply apply it to the interpretation of the substitution σ after unboxing. Notice that there we do not have to worry about idempotency of $\Box$ as in previous sections, as unbox in Cocon is already naturally modeled by unbox in dependent intuitionistic K.

The interpretation of contextual objects directly forward to the interpretation of domain-level terms. Contextual types are interpreted to dependent functions.

For computation-level types, we interpret boxed contextual types to boxed types and function types in Cocon to function types in dependent intuitionistic K as typically done. Notice that we interpret function types on both the domain level and the computation level to functions in dependent intuitionistic K, so that we have a unified syntax. It is worth mentioning that `ctx` is interpreted to $\Box\mathsf{Ctx}$, because context variables are always global in Cocon. That is why we have an additional `unbox` when interpreting contextual variables.

The interpretations of computation-level contexts and terms are immediate.

Based on this interpretation, the domain and the computation levels reside separately in two "zones" in dependent intuitionistic K. These two "zones" can be distinguished by checking whether a 🔒 exists in the context. If there is, then the current term is on the computation level, and otherwise it is on the domain level. A 🔒 is added when a `box` is encountered. This corresponds to getting into the domain level from the computation one. Therefore, dependent intuitionistic K and Cocon do seem to correspond nicely (except that the former does not have recursors for HOAS).

8 CONCLUSION

We have given a rational reconstruction of contextual type theory in presheaf models of higher-order abstract syntax. This provides a semantical way of understanding the invariants of contextual types independently of the algorithmic details of type checking. At the same time, we identify the contextual modal type theory, Cocon, which is known to be normalizing, as a syntax for presheaf models of HOAS. By accounting for the Yoneda embedding with a universe á la Tarski, we obtain a manageable way of constructing contextual types in the model, especially in the dependent case. Presheaves over models of dependent types have been used in the context of two-level type theories for homotopy type theory [Annenkov et al. 2017; Capriotti 2016]. Clarifying the precise relationship to this line of research is an interesting direction that will however require further work.

In future work, one may consider using the model as a way of compiling contextual types, by implementing the semantics. In another direction, it may be interesting to apply the syntax of contextual types to other presheaf categories. We also hope that the model will help to guide the further development of Cocon.

ACKNOWLEDGEMENT

This work was funded by the Natural Sciences and Engineering Research Council of Canada (grant number 206263), Fonds de recherche du Québec - Nature et Technologies (grant number 253521), and Postgraduate Scholarship - Doctoral by the Natural Sciences and Engineering Research Council of Canada awarded to the first author.

REFERENCES

Danil Annenkov, Paolo Capriotti, and Nicolai Kraus. 2017. Two-Level Type Theory and Applications. *ArXiv e-prints* (may 2017). http://arxiv.org/abs/1705.03307

Andrew Barber and Gordon Plotkin. 1996. *Dual intuitionistic linear logic.* University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science.

P. N. Benton, Gavin M. Bierman, Valeria de Paiva, and Martin Hyland. 1993. A Term Calculus for Intuitionistic Linear Logic. In *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*, Marc Bezem and Jan Friso Groote (Eds.), Vol. 664. Springer, 75–90.

Lars Birkedal, Ranald Clouston, Bassel Mannaa, Rasmus Ejlers Møgelberg, Andrew M. Pitts, and Bas Spitters. 2020. Modal dependent type theory and dependent right adjoints. *Math. Struct. Comput. Sci.* 30, 2 (2020), 118–138. https://doi.org/10.1017/S0960129519000197

Paolo Capriotti. 2016. *Models of type theory with strict equality.* Ph.D. Dissertation. School of Computer Science, University of Nottingham, Nottingham, UK. http://arxiv.org/abs/1702.04912

Pierre Clairambault and Peter Dybjer. 2014. The biequivalence of locally cartesian closed categories and Martin-L öf type theories. *Math. Struct. Comput. Sci.* 24, 6 (2014). https://doi.org/10.1017/S0960129513000881

Rowan Davies and Frank Pfenning. 2001. A modal analysis of staged computation. *J. ACM* 48, 3 (2001), 555–604. https://doi.org/10.1145/382780.382785

Peter Dybjer. 1995. Internal Type Theory. In *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers.* 120–134. https://doi.org/10.1007/3-540-61780-9_66

M. Fiore, G. D. Plotkin, and D. Turi. 1999. Abstract Syntax and Variable Binding. In *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, G. Longo (Ed.). IEEE Computer Society Press, 193–202.

Murdoch Gabbay and Andrew Pitts. 1999. A New Approach to Abstract Syntax Involving Binders. In *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, G. Longo (Ed.). IEEE Computer Society Press, 214–224. citeseer.ist.psu.edu/gabbay99new.html

Murdoch J. Gabbay and Aleksandar Nanevski. 2013. Denotation of contextual modal type theory (CMTT): Syntax and meta-programming. *Journal of Applied Logic* 11, 1 (March 2013), 1–29. https://doi.org/10.1016/j.jal.2012.07.002

Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. 2019. Implementing a modal dependent type theory. *Proc. ACM Program. Lang.* 3, ICFP (2019), 107:1–107:29. https://doi.org/10.1145/3341711

Robert Harper, Furio Honsell, and Gordon Plotkin. 1993. A Framework for Defining Logics. *Journal of the ACM* 40, 1 (January 1993), 143–184.

Martin Hofmann. 1997. *Syntax and Semantics of Dependent Types.* Cambridge University Press, 79–130.

Martin Hofmann. 1999. Semantical Analysis of Higher-Order Abstract Syntax. In *14th Annual IEEE Symposium on Logic in Computer Science (LICS'99)*. IEEE Computer Society, 204–213.

Bart Jacobs. 1993. Comprehension Categories and the Semantics of Type Dependency. *Theor. Comput. Sci.* 107, 2 (1993), 169–207.

G. A. Kavvos. 2017. Intensionality, Intensional Recursion, and the Gödel-Löb axiom. *CoRR* abs/1703.01288 (2017). arXiv:1703.01288 http://arxiv.org/abs/1703.01288

Daniel R. Licata, Ian Orton, Andrew M. Pitts, and Bas Spitters. 2018. Internal Universes in Models of Homotopy Type Theory. In *Formal Structures for Computation and Deduction (FSCD'18)*. 22:1–22:17.

Dale Miller and Catuscia Palamidessi. 1999. Foundational Aspects of Syntax. *ACM Comput. Surv.* 31, 3es (Sept. 1999). https://doi.org/10.1145/333580.333590

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Transactions on Computational Logic* 9, 3 (2008), 1–49.

Frank Pfenning and Conal Elliott. 1988. Higher-Order Abstract Syntax. In *ACM SIGPLAN Symposium on Language Design and Implementation (PLDI'88)*. 199–208.

Brigitte Pientka. 2008. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*. ACM Press, 371–382.

Brigitte Pientka and Andreas Abel. 2015. Structural Recursion over Contextual Objects. In *13th International Conference on Typed Lambda Calculi and Applications (TLCA'15)*, Thorsten Altenkirch (Ed.). Leibniz International Proceedings in Informatics (LIPIcs) of Schloss Dagstuhl, 273–287.

Brigitte Pientka and Joshua Dunfield. 2008. Programming with proofs and explicit contexts. In *ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*. ACM Press, 163–173.

Brigitte Pientka and Ulrich Schöpp. 2020. Semantical Analysis of Contextual Types. In *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12077)*, Jean Goubault-Larrecq and Barbara König (Eds.). Springer, 502–521. https://doi.org/10.1007/978-3-030-45231-5_26

Brigitte Pientka, David Thibodeau, Andreas Abel, Francisco Ferreira, and Rébecca Zucchini. 2019. A Type Theory for Defining Logics and Proofs. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*. IEEE, 1–13. https://doi.org/10.1109/LICS.2019.8785683

Andrew M. Pitts. 1987. Polymorphism is Set Theoretic, Constructively. In *Category Theory and Computer Science, Edinburgh, UK, September 7-9, 1987, Proceedings (Lecture Notes in Computer Science, Vol. 283)*, David H. Pitt, Axel Poigné, and David E. Rydeheard (Eds.). Springer, 12–39. https://doi.org/10.1007/3-540-18508-9_18

Michael Shulman. 2018. Brouwer's fixed-point theorem in real-cohesive homotopy type theory. *Mathematical Structures in Computer Science* 28, 6 (2018), 856–941.