The Fire Triangle
How To Mix Substitution, Dependent Elimination and Effects

Pierre-Marie Pédrot Nicolas Tabareau
Inria, Nantes
firstname.surname@inria.fr

***Abstract*—There is a critical tension between substitution, dependent elimination and effects in type theory. In this paper, we crystallize this tension in the form of a no-go theorem that constitutes the fire triangle of type theory. To release this tension, we propose ∂CBPV, an extension of call-by-push-value (CBPV) —a general calculus of effects—to dependent types. Then, by extending to ∂CBPV the well-known decompositions of call-by-name and call-by-value into CBPV, we show why, in presence of effects, dependent elimination must be restricted in call-by-name, and substitution must be restricted in call-by-value. To justify ∂CBPV and show that it is general enough to interpret many kinds of effects, we define various effectful syntactic translations from ∂CBPV to Martin-Löf type theory: the reader, weaning and forcing translations.**

I. Introduction

The addition of effects to a logical system via syntactic translations is not new and can be traced back to double-negation translations [1], although the modern standpoint can undoubtedly be attributed to Moggi in his seminal paper [2].

Since the inception of dependent type theory, several people tried to apply the techniques coming from simply-typed settings to enrich it with new reasonning principles, typically classical logic. The early attempts were mixed, if not outright failures. Most notably, Barthe and Uustalu showed that writing a typed CPS translation preserving dependent elimination was out of reach [3], and similarly Herbelin proved that CIC was inconsistent with computational classical logic [4].

Retrospectively, this should not have been that surprising. This incompatibility is the reflect of a very ancient issue: mixing of classical logic with the axiom of choice, whose intuitionistic version is a consequence of dependent elimination, is a well-known source of foundational problems [5]. While in the literature much emphasis has been put on the particular case of classical logic, we argue in this paper that this is an instance of a broader phenomenon, namely that side-effects are at odds with dependent type theory, in a pick two out of three conundrum. This mismatch is evocatively dubbed the Fire Triangle (Fig. 1) and is discussed in detail in Section II.

To get out of this pit, we propose in this article a generic solution based on well-known tools coming from the study of the semantics of programming languages, allowing to safely add effects to type theory. It consists in a generalization of Levy's CBPV [6] to the dependently-typed setting, whose design has been fueled by the recent work of the authors' on effectful type theories. In particular, we provide syntactic models of this system to justify it, as well as decompositions into it arising from the usual embeddings into CBPV.

978-1-5090-3018-7/17/$31.00

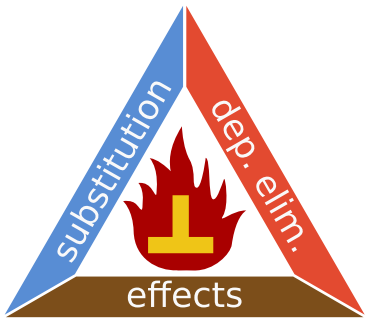
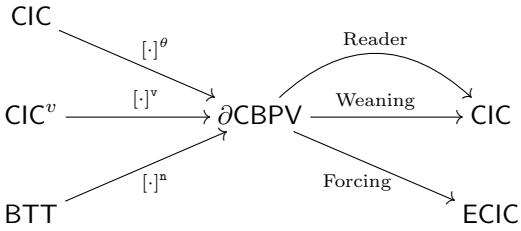


Fig. 1. The substitution-dependent elimination-effects triangle

Plan of the paper.

In Section II, we dive into the fundamental impossibility theorem that explains why people had a hard time extending type theory with effects. This is the major insight of the paper and will be used to give intuitions about the workarounds.

Section III describes ∂CBPV, a system that allows to safely mix effects and dependent type theory. In Sections IV, V and VI, we give syntactic decompositions from several flavours of type theory into ∂CBPV. Dually, Sections VII, VIII and IX provide syntactic models of ∂CBPV based on previous work. All of these translations are summarized bellow.



Finally, Section X provides further comparisons with similar attempts.

II. Substitution, Dependent Elimination and Effects

A. The Fire Triangle: A general no-go theorem

Let us now make precise this tension between substitution, dependent elimination and effects. To formulate and prove a general no-go theorem, we first need to make formal what

we mean by each of the three notions under consideration. To remain as abstract as possible with respect to the underlying type theory, we will use to different typing judgments:

$$\Gamma \vdash u : A$$

saying that t has type A in context Γ and

$$\Gamma \vdash A$$

saying that A is valid (or inhabited) in context Γ.

In this setting, we can readily express what it means for a type theory to feature substitution: a valid type containing a free variable $x : A$ is still valid when x is substituted with any term $u : A$.

Definition 1 (Substitution)**.** A type theory enjoys *substitution* if the following rule is admissible.

$$\frac{\Gamma, x : A \vdash B \qquad \Gamma \vdash u : A}{\Gamma \vdash B\{x := u\}}$$

To express what it means to feature dependent elimination and effects, we need to consider a basic type with at least two elements. We thus now assume that the theory features a type $\mathbb{B}$ with two inhabitants $\vdash \texttt{true} : \mathbb{B}$ and $\vdash \texttt{false} : \mathbb{B}$. In this setting, dependent elimination on booleans can simply be stated as the fact that if a type with one boolean free variable x is valid when x is substituted by `true` and by `false`, then it is valid in general.

Definition 2 (Dependent elimination)**.** A type theory enjoys *dependent elimination* on booleans if the following rule is admissible (where $\square$ denotes a universe of types).

$$\frac{\Gamma, x : \mathbb{B} \vdash A : \square \qquad \Gamma \vdash A\{x := \texttt{true}\} \qquad \Gamma \vdash A\{x := \texttt{false}\}}{\Gamma, x : \mathbb{B} \vdash A}$$

Dependent elimination can be generalized to all inductive types, and is the type theory equivalent to induction principles.

Finally, we need to express what it means for a type theory (or programming language) to be effectful. Intuitively, a type theory is pure when every term observationally behaves as a value. So a simple way to formalize what it means to be effectful is to say that there exists a boolean term which is not observationally equivalent to `true` nor `false`.

Definition 3 (Effects)**.** A type theory is *effectful* if there exists a closed term $\vdash t : \mathbb{B}$ that is not observationally equivalent to a value, that is, there exists a context C such that $C[\texttt{true}] \equiv \texttt{true}$ and $C[\texttt{false}] \equiv \texttt{true}$, but $C[t] \equiv \texttt{false}$ (where $\equiv$ denotes definitional equality).

With those three notions in hand, we can state and prove a generalization of Herbelin's paradox, which is actually pointing out its essence, and provide a no-go theorem for a type theory featuring at the same time substitution, dependent elimination and effects. In the following, we assume that $\bot$ is the empty type and $\top$ the type with exactly one element.

Theorem 1 (Fire triangle)**.** *An effectful type theory that enjoys substitution and dependent elimination is inconsistent.*

Proof. We define (Leibniz) equality by

$$t = u := \Pi P : A \to \square.\, P\ t \leftrightarrow P\ u.$$

Note that we could equivalently assume that the type theory features identity types. We take t and C as provided by Definition 2. By dependent elimination, it holds that $x : \mathbb{B} \vdash C[x] = \texttt{true}$. By substitution, $\vdash C[t] = \texttt{true}$. By conversion and because $C[t] \equiv \texttt{false}$, this implies $\vdash \texttt{false} = \texttt{true}$.

But, by dependent elimination, we also have $\vdash \texttt{false} = \texttt{true} \to \bot$. Indeed, instanting $\texttt{false} = \texttt{true}$ with P defined by $P\ \texttt{false} \equiv \bot$ and $P\ \texttt{true} \equiv \top$, we get an inhabitant of $\bot$ from an inhabitant of $\top$. $\square$

Example 1. An archetypical example of an effectful term can be obtained with `callcc` [7]. It is indeed possible to use it to write a term $\texttt{decide} : \square \to \mathbb{B}$ that decides whether a type is inhabited. Obviously, $\texttt{decide}\ A$ cannot enjoy canonicity in general. Such booleans are called *backtracking* or *non-standard*, and are the root of Herbelin's paradox.

Before looking at a way to tame this fire triangle, let us look at the consequence of this theorem when the evaluation strategy is fixed—either call-by-value or call-by-name.

B. Substitution in call-by-value.

The by-value β-reduction is the congruence closure of the generator

$$(\lambda x : A.\, t)\ v \to_{\texttt{v}} t\{x := v\}$$

where v is a syntactic value. As every function in call-by-value can expect its argument to be a value, this explains why dependent elimination as defined in Definition 1 is always valid: every predicate on $\mathbb{B}$ holds as soon as it holds on `true` and `false`, because they are the only non-variable values of that type.

Constrastingly, substitution cannot hold in general, because it would imply that if a type with an open (boolean) variable is valid, it remains valid when the variable is substituted by *any* term. This is not correct if there are effectful terms, making the substitution by an arbitrary term invalid. This explains why, in a call-by-value setting, one usually consider a *value restriction* [8], [9] when a substitution is involved, *e.g.,*

$$\frac{\Gamma \vdash v : A \qquad \Gamma, x : A \vdash B : \square_i \qquad \Gamma, x : A \vdash u : B}{\Gamma \vdash \texttt{let}\ x : A := v\ \texttt{in}\ u : B\{x := v\}}$$

where v is required to be a syntactic value.

C. Dependent elimination in call-by-name.

The by-name β-reduction is the congruence closure of the generator

$$(\lambda x : A.\, t)\ u \to_{\texttt{n}} t\{x := u\}$$

where u is any term of the type theory. This means that in a call-by-name setting, substitution always holds by construction. However, as already noticed in [10], [11], dependent elimination is now lost in general. Clearly, if there are effectful terms, knowing the behaviour of a predicate on boolean *values*

value types	$A, B ::= \mathcal{U}\ X \mid \mathbb{B}$
computation types	$X, Y ::= A \to X \mid \mathcal{F}\ A$

Fig. 2. Call-by-push-value (types only)

is not enough to know the behaviour of the predicate in general. Intuitively, this is because doing a case analysis on a boolean term triggers the evaluation of the term into a value, thus potentially performing some effects. If this evaluation is not triggered also in the type, there is a kind of desynchronization between effects performed in the term and effects performed in the type.

To recover consistency of the theory, one may consider Baclofen Type Theory [11] (BTT) which is a way to enforce this synchronization. More specifically, on boolean terms, one needs to provide first non-dependent case analysis

$$\mathtt{rec}_{\mathbb{B}} : \Pi P : \square.\ P \to P \to \mathbb{B} \to P.$$

Using case analysis, it is possible to define of boolean storage operator $\sigma_{\mathbb{B}}$ which takes a boolean predicate and returns another similar predicate that starts by doing a case analysis on its argument.

$$\begin{array}{lcl} \sigma_{\mathbb{B}} & : & \mathbb{B} \to (\mathbb{B} \to \square) \to \square \\ & := & \mathtt{rec}_{\mathbb{B}}\ ((\mathbb{B} \to \square) \to \square)\ (\lambda k.\, k\ \mathtt{true})\ (\lambda k.\, k\ \mathtt{false}) \end{array}$$

This notion of storage operator has been introduced by Krivine in classical realisability [12] to solve fundamentally the same problem, that is, implementing induction over classical integers. Using this storage operator, it is now possible to define a dependent case analysis which reflects the triggered evaluation in the term by the use of the storage operator in the type.

$$\mathtt{drec}_{\mathbb{B}} : \Pi P : \mathbb{B} \to \square.\ P\ \mathtt{true} \to P\ \mathtt{false} \to \Pi b : \mathbb{B}.\, \sigma_{\mathbb{B}}\ b\ P$$

D. Examples

In the literature, there are to our knowledge four ways of dealing with this fire triangle:

1) No effects + substitution + dependent elimination: this is the good old plain CIC.
2) Effects + dependent elimination + restricted substitution: albeit not strictly speaking dependent type theory, this is the path followed by PML [9].
3) Effects + substitution + restricted dependent elimination: this is what BTT [11] is all about.
4) Effects + substitution + dependent elimination, but inconsistent: the exceptional type theory [13] is an instance of this. One can argue that this is a paradigm shift from a dependent *type theory* to a dependently-typed *programming language*, where consistency is not relevant.

Let us now turn to a subsuming approach, by lifting a well-established calculus of effects to a dependent setting.

E. Explicit handling of effects: call-by-push-value

Call-by-push-value [6] (CBPV) was introduced by Levy to provide a unified setting in which to talk about call-by-name and call-by-value evaluations. It clarifies the situation by describing both call-by-value and call-by-name as two distinct embeddings, leading to a more atomic presentation. CBPV's types (and terms) are divided into two classes: pure values and effectful computations (see Figure 2). It is possible to go from one to the other using the two type constructors $\mathcal{U}$ and $\mathcal{F}$ that mimic the two parts of the adjunction decomposing a computational monad. In this presentation, the function space is a computation type, with domain a value type and data type (such as booleans) are value type. Call-by-name and call-by-value strategies can then be decomposed into CBPV.

The by-value translation $[-]^{\mathtt{v}}$ is defined on arrows as

$$[A \to B]^{\mathtt{v}} \quad := \quad \mathcal{U}\ ([A]^{\mathtt{v}} \to \mathcal{F}\ [B]^{\mathtt{v}})$$

and the correctness lemma states that when $\Gamma \vdash t : A$ then

$$[\Gamma]^{\mathtt{v}} \vdash_c [t]^{\mathtt{v}} : \mathcal{F}\ [A]^{\mathtt{v}}.$$

Here, the need for value restriction in substitution appears clearly because a variable of type A is translated as a variable of the value type $[A]^{\mathtt{v}}$, whereas a term of type A is translated as a term of the computation type $\mathcal{F}\ [A]^{\mathtt{v}}$. Therefore, not every term can substitute a variable, only those that corresponds to a value. Note that the value restriction is a syntactic notion, but in CBPV, it is possible to express a semantic notion of being as a value, called *thunkability*.

The by-name translation $[-]^{\mathtt{n}}$ is defined as

$$[A \to B]^{\mathtt{n}} \quad := \quad \mathcal{U}\ [A]^{\mathtt{n}} \to [B]^{\mathtt{n}}.$$

and the correctness lemma states that when $\Gamma \vdash t : A$ then

$$\mathcal{U}\ [\Gamma]^{\mathtt{n}} \vdash_c [t]^{\mathtt{n}} : [A]^{\mathtt{n}}.$$

Here, substitution is always valid as a variable of type A is translated as a variable of the value type $\mathcal{U}\ [A]^{\mathtt{n}}$, whereas a term of type A is translated as a term of the computation type $[A]^{\mathtt{n}}$. Thus any (thunk of a) term can substitute a variable. However, the translation of $\mathbb{B}$ is given by $\mathcal{F}\ \mathbb{B}$ and elimination is encoded by first evaluating the term into a boolean value and then applying the elimination principle. This is the reason why in call-by-name, this implicit evaluation performed by dependent elimination has to be reflected in the type, giving rise to BTT. Note that dually to thunkability, there is a more semantic version of storage operators, which caracterizes which predicates morally starts by evaluating their argument. The semantic property is called linearity, a notion that has been first described by Munch-Maccagnoni [14] and rephrased recently in the context of CBPV by Levy [15] (see Section III-E for a definition of thunkability and linearity).

We advocate in this paper that providing a good definition of a dependent version of CBPV, dubbed ∂CBPV, is the key to understanding the interaction between substitution, dependent elimination and effects.

F. Taming the fire triangle: dependent call-by-push-value.

Several attempts have already been performed to define a dependent version of CBPV. But to do this, one need to solve one main issue:

"How to define a dependent version of the `let` *binder?"*

Indeed, the introduction rule for `let` in CBPV is given by the following rule:

$$\frac{\Gamma \vdash_c t : \mathcal{F}\ A \qquad \Gamma, x : A \vdash_c u : X}{\Gamma \vdash_c \texttt{let}\ x : A := t\ \texttt{in}\ u : X}$$

But if we assume that X depends on $x : A$, it is not possible to directly substitute x for t because t has type $\mathcal{F}\ A$. In [16], this problem has been solved by considering a value restriction, similarly to what is done to solve a similar issue in call-by-value. But we advocate here for a more general solution, which corresponds more closely to the solution introduced in BTT: using a `let` binder also in the type to synchronize the evaluation of t in the term and in the type.

However, we cannot simply introduce an introduction rule for `let` where `let` appears on both side. Indeed, doing this, we would not access to the non-dependent `let` anymore as in general $\texttt{let}\ x : A := t\ \texttt{in}\ u$ is not convertible to u even if u does not depend on x. Intuitively, this is because the first term performs the effects present in t while the second one does not.

The other central question that needs to be solved to turn CBPV into a proper dependent type theory is:

"What is the notion of universes in presence of effects?"

Indeed, one may wonder whether a universe of types deals with value types or computation types and whether it is itself a value type or a computation type? In [17], Ahman introduces only a universe of value types, which is itself a value type. But then, to prevent this universe from being trivial, he has to define a value-typed version of dependent product in the theory. Not only this departs from the standard definition of CBPV, but we also claim that this turns out to be a *faux pas* preventing to talk about pervasive and crucial structure arising from our models. In this paper, we advocate that the notion of universes in ∂CBPV should reflect the structure of syntax, with a universe hierarchy $\square_i^v$ of value types and an orthogonal universe hierarchy $\square_i^c$ of computation types. In this setting, $\mathcal{F}$ can be seen as a function from $\square_i^v$ to $\square_i^c$ and $\mathcal{U}$ as a function in the backward direction—making $\square_i^v$ and $\square_i^c$ interact. But as universes are themselves types, one may wonder whether $\mathcal{F}\ \square_i^v$ is convertible to $\square_i^c$, and dually whether $\mathcal{U}\ \square_I^c$ is convertible to $\square_i^v$.

This suggests that finding the right presentation and equational theory for ∂CBPV is not an easy matter. In this paper, we depart from the usual categorical model approach and choose to use a more syntactic guideline, looking at effectful program transformations already existing in the literature.

G. A syntactic guideline: weaning and forcing.

We follow the general approach of syntactic models advocated for in [18]. Recall that the base idea is to show the consistency of a source theory $\mathcal{S}$ using a translation into a target theory $\mathcal{T}$, for which we already know consistency. Technically, this amounts to any term M in $\mathcal{S}$ being translated by induction over its syntax into a term $[M]$ in $\mathcal{T}$, through a *typing soundness* theorem stating that $\mathsf{El}\ [\Gamma] \vdash [M] : \mathsf{El}\ [A]$ whenever $\Gamma \vdash M : A$. Here, El is an internal operation which coerces a translated type into a type of the target type theory.

In this paper, the critical point is not consistency, which can be simply proven by translating ∂CBPV directly to Martin-Löf type theory (MLTT) or the Calculus of Inductive Constructions (CIC)[1], interpreting every effectful operator trivially, thus giving a pure model of ∂CBPV. Rather, we focus more on an auxiliary lemma used to prove typing soundness: a form of *computational soundness* which says that $[M] \equiv [N]$ whenever $M \equiv N$. More precisely, syntactical models provide a meaning interpretation of effectful operations, which makes it possible to distinguish the right equational theory for ∂CBPV shared by all such models.

There are two main effectful program transformation that have been considered into CIC: (i) the forcing translation with its call-by-value [19] and call-by-name [10] variants, (ii) the weaning translation [11] which corresponds to a call-by-name variant of Moggi's monadic translation. Those two translations provide two extreme points in the possible syntactical models of CBPV, where either $\mathcal{F}$ or $\mathcal{U}$ is degenerated. A significant part of this paper is to show that those translations can be extended to translations from ∂CBPV to CIC.

III. Dependent Call-By-Push-Value

In this section, we present an extension of Levy's CBPV [6] to dependent types. We coined the name ∂CBPV not to confuse it with Vákár's dCBPV.

A. Syntax of ∂CBPV.

As usual, ∂CBPV's types and terms are divided into two classes: pure values v and effectful computations t, a dichotomy which is reflected in the typing rules. Note that contrarily to the simply typed setting, we can not distinguish terms and types anymore. To ease the reading, we do not use the usual underline notion for computations, and rather use the convention that capital letters of the beginning of the latin alphabet $(A, B, \dots)$ are for value types and capital letters of the end of the latin alphabet $(X, Y, \dots)$ are for computation types. The syntax and typing rules are given at Figure 3. The terms are given by their typing judgement, written $\Gamma \vdash_v v : A$ for values and $\Gamma \vdash_c t : X$ for computations, where Γ is a context of values, that is a finite sequence $x_0 : A_0; \dots; x_n : A_n$ of identifiers associated to a *value* type. Note that the annotation specifying the kind of sequent at hand is only written out out for readability, as the the separation between values and computations is enforced by typing.

B. Meaning of types.

As we have said, there are two classes of types, value types and computation types. Those two classes are reflecting respectively by two parallel universe hierarchies $\square_i^v$ and $\square_i^c$. Note that the typing judgment $\Gamma \vdash_v v : A$ implies that A is a value type of sort $\square_i^v$ for some i, and similarly for

[1]We do not make a strong distinction between MLTT and CIC, as we consider CIC without the universe of propositions, which is very similar to MLTT. In the sequel, we refer to CIC for this common setting.

$$
\begin{array}{lll}
\text{values} & A, B, v, w ::= \square_i^v \mid \mathcal{U}\ X \mid \Sigma x : A.\, B \mid A + B \mid \mathtt{eq}\ A\ v\ w \mid x \mid \mathtt{thunk}\ t \mid (v, w) \mid \mathtt{inl}\ v \mid \mathtt{inr}\ w \mid \mathtt{refl} \\
\text{computations} & X, Y, t, u ::= \square_i^c \mid \mathcal{F}\ A \mid \Pi x : A.\, X \mid \mathtt{force}\ t \mid \lambda x : A.\, t \mid t\ v \mid \mathtt{let}\ x : A := t\ \mathtt{in}\ u \mid \mathtt{return}\ v \\
& \qquad \mid \mathtt{rec}_\Sigma(v, X, t) \mid \mathtt{rec}_+(v, X, t_1, t_2) \mid \mathtt{rec}_{\mathtt{eq}}(v, X, t) \\
\text{environments} & \Gamma ::= \cdot \mid \Gamma, x : A
\end{array}
$$

$$\frac{}{\vdash \cdot} \qquad \frac{\Gamma \vdash_v A : \square_i^v}{\vdash \Gamma, x : A} \qquad \frac{\Gamma \vdash_v A : \square_i^v}{\Gamma, x : A \vdash_v x : A} \qquad \frac{\Gamma \vdash_v B : \square_i^v \qquad \Gamma \vdash_v x : A}{\Gamma, y : B \vdash_v x : A}$$

$$\frac{\Gamma \vdash_c t : Y \qquad X \equiv Y \qquad \Gamma \vdash_c X : \square_i^c}{\Gamma \vdash_c t : X} \qquad \frac{\Gamma \vdash_v v : B \qquad A \equiv B \qquad \Gamma \vdash_v A : \square_i^v}{\Gamma \vdash_v v : A}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash_v \square_i^v : \square_{i+1}^v} \qquad \frac{\vdash \Gamma}{\Gamma \vdash_c \square_i^c : \square_{i+1}^c} \qquad \frac{\Gamma \vdash_v A : \square_i^v}{\Gamma \vdash_c \mathcal{F}\ A : \square_i^c} \qquad \frac{\Gamma \vdash_c X : \square_i^c}{\Gamma \vdash_v \mathcal{U}\ X : \square_i^v} \qquad \frac{\Gamma \vdash_v A : \square_i^v \qquad \Gamma \vdash_v v : A \qquad \Gamma \vdash_v w : A}{\Gamma \vdash_v \mathtt{eq}\ A\ v\ w : \square_i^v}$$

$$\frac{\Gamma \vdash_v A : \square_i^v \qquad \Gamma, x : A \vdash_c X : \square_j^c}{\Gamma \vdash_c \Pi x : A.\, X : \square_{\max(i,j)}^c} \qquad \frac{\Gamma \vdash_v A : \square_i^v \qquad \Gamma, x : A \vdash_v B : \square_j^v}{\Gamma \vdash_v \Sigma x : A.\, B : \square_{\max(i,j)}^v} \qquad \frac{\Gamma \vdash_v A : \square_i^v \qquad \Gamma \vdash_v B : \square_j^v}{\Gamma \vdash_v A + B : \square_{\max(i,j)}^v}$$

$$\frac{\Gamma \vdash_c t : X}{\Gamma \vdash_v \mathtt{thunk}\ t : \mathcal{U}\ X} \qquad \frac{\Gamma \vdash_v v : \mathcal{U}\ X}{\Gamma \vdash_c \mathtt{force}\ v : X} \qquad \frac{\Gamma \vdash_v v : A}{\Gamma \vdash_c \mathtt{return}\ v : \mathcal{F}\ A}$$

$$\frac{\Gamma \vdash_c t : \mathcal{F}\ A \qquad \Gamma \vdash_c X : \square_i^c \qquad \Gamma, x : A \vdash_c u : X}{\Gamma \vdash_c \mathtt{let}\ x : A := t\ \mathtt{in}\ u : X} \qquad \frac{\Gamma, x : A \vdash_c X : \square_i^c \qquad \Gamma, x : A \vdash_c t : X}{\Gamma \vdash_c \lambda x : A.\, t : \Pi x : A.\, X}$$

$$\frac{\Gamma \vdash_c t : \mathcal{F}\ A \qquad \Gamma, x : A \vdash_c X : \square_i^c \qquad \Gamma, x : A \vdash_c u : X}{\Gamma \vdash_c \mathtt{dlet}\ x : A := t\ \mathtt{in}\ u : \mathtt{let}\ x : A := t\ \mathtt{in}\ X} \qquad \frac{\Gamma \vdash_c t : \Pi x : A.\, X \qquad \Gamma \vdash_v v : A}{\Gamma \vdash_c t\ v : X\{x := v\}}$$

$$\frac{\Gamma \vdash_v v : A \qquad \Gamma \vdash_v w : B\{x := v\} \qquad \Gamma \vdash_v \Sigma x : A.\, B : \square_i^v}{\Gamma \vdash_v (v, w) : \Sigma x : A.\, B}$$

$$\frac{\Gamma \vdash_v v : \Sigma x : A.\, B \qquad \Gamma \vdash_c X : (\Sigma x : A.\, B) \to \square_i^c \qquad \Gamma \vdash_c t : \Pi (x : A)\, (y : B).\, X\ (x, y)}{\Gamma \vdash_c \mathtt{rec}_\Sigma(v, X, t) : X\ v}$$

$$\frac{\Gamma \vdash_v A : \square_i^v \qquad \Gamma \vdash_v B : \square_i^v \qquad \Gamma \vdash_v v : A}{\Gamma \vdash_v \mathtt{inl}\ v : A + B} \qquad \frac{\Gamma \vdash_v A : \square_i^v \qquad \Gamma \vdash_v B : \square_i^v \qquad \Gamma \vdash_v w : B}{\Gamma \vdash_v \mathtt{inr}\ w : A + B}$$

$$\frac{\Gamma \vdash_v v : A + B \qquad \Gamma \vdash_c X : (A + B) \to \square_i^c \qquad \Gamma \vdash_c u_1 : \Pi x : A.\, X\ (\mathtt{inl}\ x) \qquad \Gamma \vdash_c u_2 : \Pi y : B.\, X\ (\mathtt{inr}\ y)}{\Gamma \vdash_c \mathtt{rec}_+(v, X, u_1, u_2) : X\ v}$$

$$\frac{\Gamma \vdash_v A : \square_i^v \qquad \Gamma \vdash_v v : A \qquad \Gamma \vdash_v w : A}{\Gamma \vdash_v \mathtt{eq}\ A\ v\ w : \square_i^v} \qquad \frac{\Gamma \vdash_v A : \square_i^v \qquad \Gamma \vdash_v v : A}{\Gamma \vdash_v \mathtt{refl} : \mathtt{eq}\ A\ v\ v}$$

$$\frac{\Gamma \vdash_v v : \mathtt{eq}\ A\ w_1\ w_2 \qquad \Gamma \vdash_c X : \Pi y : A.\, \mathtt{eq}\ A\ w_1\ y \to \square_i^c \qquad \Gamma \vdash_c t : X\ w_1\ \mathtt{refl}}{\Gamma \vdash_c \mathtt{rec}_{\mathtt{eq}}(v, X, t) : X\ w_2\ v}$$

Fig. 3. Dependent call-by-push-value

$\Gamma \vdash_c t : X$ and $\square_i^c$. In particular, value (resp. computation) types are always values (resp. computations).

Those hierarchies are parallel in the sense that $\square_i^v$ has type $\square_{i+1}^v$ and $\square_i^c$ has type $\square_{i+1}^c$. But the are also connected by two operations: $\mathcal{F}$ which transforms a value type into a computation type and $\mathcal{U}$ which transforms a computation type into a value type. For simplicity of the presentation of the system, we do not consider more refined notions on the universe hierarchies such as cumulativity or universe polymorphism but they can be integrated smoothly as those notions are largely independent from the notion of value and computation types.

As for function types in CBPV, dependent products of the form $\Pi x : A.\, X$ are computation types, with domain a value type and codomain a computation type. Note that the fact that the domain is a value type is necessary because contexts are only composed of value types. As it is the case in CIC, to preserve the stratification induced by the universe hierarchy, the universe level of the dependent product is the maximum of the universe levels of its domain and codomain.

Contrastingly, inductive types are value types whose type parameters are also value types. Here, we only consider three representative instances, namely dependent sums $\Sigma x : A.\, B$, coproducts $A + B$ and equality $\mathtt{eq}\ A\ v_1\ v_2$.

C. Meaning of terms.

Let us first recall the intuition behind the terms coming directly from CBPV. The `thunk` primitive is to be understood as a way of *boxing* a computation into a value. Its dual `force`

runs the computation.[2] The `return` primitive creates a pure computation from a value. The (non-dependent) `let` binding first evaluates its argument, possibly generating some effects, binds the purified result to the variable and continues with the remaining term.

Dependent products come as usual with a notion of λ-abstraction and application. The rule for application $t\ v$ performs directly the substitution in the (dependent) type, without any restriction, because v is already a value.

The main addition is a dependent version of the `let` binding, that we call `dlet`. It behaves as `let` but the type of the conclusion cannot by X anymore, nor a direct substitution $X\{x := t\}$ because t is not a value. This is why we need also to evaluate t in the type, meaning that the type of the conclusion is a (non-dependent) `let` itself. Note that we cannot use a single rule for both dependent and non-dependent `let` binding, contrarily to what happens for dependent product and arrow type. This is because if we do so, $\texttt{let}\ x : A := t\ \texttt{in}\ X$ would have type $\texttt{let}\ x : A := t\ \texttt{in}\ \square_i^c$ which is not a sort of the system. Indeed, as we will see using syntactical models such as forcing or weaning, the rule

$$\texttt{let}\ x : A := t\ \texttt{in}\ \square_i^c \equiv \square_i^c$$

is not admissible in ∂CBPV because the left-hand side performs the effects of t while the right-hand side does not. We insist that it is critical for $\square_i^c$ to be itself a computation, for otherwise the type of `dlet` would not make sense. This is a major difference with Ahman's system [17].

Finally, inductive types comes with their usual constructors and (dependent) elimination rule. As for application, the substitution in the type of the conclusion can be performed directly, without any restriction, because v is a value.

D. Reduction of ∂CBPV.

Definition 4 (∂CBPV reduction)**.** We define the ∂CBPV reduction as the congruence closure of the following generators.

$$\begin{array}{lcl}
(\lambda x : A.\, t)\ v & \to & t\{x := v\} \\
\texttt{let}\ x : A := \texttt{return}\ v\ \texttt{in}\ t & \to & t\{x := v\} \\
\texttt{dlet}\ x : A := \texttt{return}\ v\ \texttt{in}\ t & \to & t\{x := v\} \\
\texttt{force}\ (\texttt{thunk}\ t) & \to & t \\
\texttt{rec}_\Sigma((v, w), X, u) & \to & u\ v\ w \\
\texttt{rec}_+(\texttt{inl}\, v, X, u_1, u_2) & \to & u_1\ v \\
\texttt{rec}_+(\texttt{inr}\, w, X, u_1, u_2) & \to & u_2\ w \\
\texttt{rec}_{\texttt{eq}}(\texttt{refl}, X, u) & \to & u
\end{array}$$

We write $\equiv$ for the equivalence generated by this reduction when the context is clear, otherwise we may subscript it as $\equiv_{\partial\mathsf{CBPV}}$.

Remark 1. We do not include the usual associativity rules for `let`-bindings and their `dlet` counterparts in ∂CBPV conversion. These rules happen to hold in our models, but in general not *definitionally*, sometimes even requiring adding the function extensionality axiom to CIC. Thus there is no hope to consider them for conversion in an intensional setting.

[2]This name has nothing to do with forcing itself and is a coincidence.

E. Unifying Thunkability and Linearity

As we have already mentioned in Section II, there are two central notions to consider when looking at a dependent version of CBPV, thunkability for substitution and linearity for large dependent elimination. Thunkability for CBPV has been introduced by Levy [6] after the work of Führmann [20]. It semantically expresses the fact that a potentially effectful computation is effect-free, *i.e.,* behaves as a value without performing any effect. Linearity has been considered by Munch-Maccagnoni [14] and rephrased recently in the context of CBPV by Levy [15]. It semantically expresses the fact that a function is effect-preserving, *e.g.,* by evaluating its arguments first and once. In the works of [14], [15], there are several equivalent definitions of those notions, which make the duality more or less explicit. However, those definitions are equivalent in the model, that is from an extensional point of view. In this paper, we work in an intensional setting, so the way definitions are formulated matters.

We base our definitions of thunkability and linearity on the following notion of compatibility between functions and effectful computations.

Definition 5 (Compatibility)**.** A function $f : \mathcal{U}\ \mathcal{F}\ A \to X$ and an effectful computation $t : \mathcal{F}\ A$ are said to be *compatible*, written $f \perp\!\!\!\perp t$, when the following definitional equation holds:

$$\texttt{let}\ x : A := t\ \texttt{in}\ f\ (\texttt{thunk}\ (\texttt{return}\ x)) \equiv f\ (\texttt{thunk}\ t).$$

From this notion of compatibility, one can recover both linearity and thunkability by focusing at a universal compatibility property of either the function or of the effectful computation.

Definition 6 (Linearity)**.** A function $f : \mathcal{U}\ \mathcal{F}\ A \to X$ is *linear* when for every effectful computation $t : \mathcal{F}\ A$, we have $f \perp\!\!\!\perp t$.

Definition 7 (Thunkability)**.** An effectful computation $t : \mathcal{F}\ A$ is *thunkable* when for every function $f : \mathcal{U}\ \mathcal{F}\ A \to X$, we have $f \perp\!\!\!\perp t$.

We show in the rest of this paper that this particular way of formulating thunkability and linearity is appropriate, when describing predicate on which substitution can be defined (in call-by-value) and predicates on which dependent elimination can be performed (in call-by-name).

IV. Call-by-Name Translation

In this section, we provide the extension to a dependent setting of the call-by-name translation of the simply-type λ-calculus into CBPV. Here, the source of the translation is not CIC, but a version with a restricted dependent elimination, that is called BTT. We discuss at the end of this section how BTT could be extended using the notion of linearity.

A. Call-by-name translation: the negative fragment

We define in this section the translation of CC_ω into ∂CBPV. The source system constitutes what is known as the negative fragment, i.e. a type theory whose only type formers are Π-types and a tower of universes. For the sake

of conciseness, we will not recall the rules of CC_ω, which are standard.

Definition 8 (By-name translation)**.** The by-name translation $[-]^{\mathtt{n}}$ from CC_ω into $\partial\mathsf{CBPV}$ is defined as follows.

$$\begin{array}{lcl}
[\Box_i]^{\mathtt{n}} & := & \Box_i^c \\
[\Pi x : A.\, B]^{\mathtt{n}} & := & \Pi x : \mathcal{U}\ [A]^{\mathtt{n}}.\, [B]^{\mathtt{n}} \\
[x]^{\mathtt{n}} & := & \mathtt{force}\ x \\
[t\ u]^{\mathtt{n}} & := & [t]^{\mathtt{n}}\ (\mathtt{thunk}\ [u]^{\mathtt{n}}) \\
[\lambda x : A.\, t]^{\mathtt{n}} & := & \lambda x : \mathcal{U}\ [A]^{\mathtt{n}}.\, [t]^{\mathtt{n}}
\end{array}$$

This translation is very similar to the call-by-name embedding of simply-typed λ-calculus into CBPV. In particular, $[A \to B]^{\mathtt{n}} := \mathcal{U}\ [A]^{\mathtt{n}} \to [B]^{\mathtt{n}}$ provided that the arrow is interpreted as a non-dependent product. Also, every CC_ω term is translated as a computation.

As expected, this translation preserve conversion and typing.

Proposition 1 (Substitution)**.** *We have*

$$[t\{x := r\}]^{\mathtt{n}} \equiv_{\partial\mathsf{CBPV}} [t]^{\mathtt{n}}\{x := \mathtt{thunk}\ [r]^{\mathtt{n}}\}.$$

Proposition 2. *If* $t \equiv_{\mathsf{CC}_\omega} u$ *then* $[t]^{\mathtt{n}} \equiv_{\partial\mathsf{CBPV}} [u]^{\mathtt{n}}$.

Proposition 3. *If* $\Gamma \vdash_{\mathsf{CC}_\omega} t : A$ *then* $\mathcal{U}\ [\Gamma]^{\mathtt{n}} \vdash_c [t]^{\mathtt{n}} : [A]^{\mathtt{n}}$.

B. Extension to BTT.

The call-by-name translation is known for being biased towards the negative fragment. Most notably, the above translation does not use the $\mathcal{F}$ type former and the associated terms at all. This contrasts with the interpretation of inductive types, which has important consequences on their interplay with effects. For the sake of conciseness, we only detail the translation of coproducts here.

Sum are translated by first lifting the translation of underlying types using $\mathcal{U}$, applying the sum and the lifting back the resulting value type to a computation type using $\mathcal{F}$.

$$[A + B]^{\mathtt{n}} := \mathcal{F}\ ((\mathcal{U}\ [A]^{\mathtt{n}}) + (\mathcal{U}\ [B]^{\mathtt{n}}))$$

The translation of constructors is analogous

$$\begin{array}{lcl}
[\mathtt{inl}\, t]^{\mathtt{n}} & := & \mathtt{return}\ (\mathtt{inl}\, (\mathtt{thunk}\ [t]^{\mathtt{n}})) \\
[\mathtt{inr}\, t]^{\mathtt{n}} & := & \mathtt{return}\ (\mathtt{inr}\, (\mathtt{thunk}\ [t]^{\mathtt{n}}))
\end{array}$$

The translation of the recursor is more problematic as it requires to recover the value out of the computation by using a `let` binder. As mentionned in Section II-C, the dependent elimination principle is restricted in BTT. For instance, for the case of coproducts, there are two recursors: the non-dependent $\mathtt{rec}_+$, and the dependent $\mathtt{drec}_+$. The former is as usual:

$$\frac{\begin{array}{ll}\Gamma \vdash_{\mathsf{BTT}} t : A + B & \Gamma \vdash_{\mathsf{BTT}} u_1 : A \to P \\ \Gamma \vdash_{\mathsf{BTT}} P : \Box_i & \Gamma \vdash_{\mathsf{BTT}} u_2 : B \to P\end{array}}{\Gamma \vdash_{\mathsf{BTT}} \mathtt{rec}_+(t, P, u_1, u_2) : P}$$

and the latter has a type guarded by a storage operator:

$$\frac{\begin{array}{ll}\Gamma \vdash_{\mathsf{BTT}} t : A + B & \Gamma \vdash_{\mathsf{BTT}} u_1 : \Pi x : A.\, P\ (\mathtt{inl}\, x) \\ \Gamma \vdash_{\mathsf{BTT}} P : (A + B) \to \Box_i & \Gamma \vdash_{\mathsf{BTT}} u_2 : \Pi y : B.\, P\ (\mathtt{inr}\, y)\end{array}}{\Gamma \vdash_{\mathsf{BTT}} \mathtt{drec}_+(t, P, u_1, u_2) : \sigma_{A+B}\ t\ P}$$

where σ_{A+B} is the storage operator defined as

$$\begin{array}{lcl}
\sigma_{A+B} & : & (A + B) \to ((A + B) \to \Box) \to \Box \\
& := & \lambda v.\, \mathtt{rec}_+\ (v,\, ((A + B) \to \Box) \to \Box, \\
& & \qquad \lambda x\, k.\, k\ (\mathtt{inl}\, x),\, \lambda y\, k.\, k\ (\mathtt{inr}\, y))
\end{array}$$

Thus, the separation between `let` and `dlet` in ∂CBPV is reflected in BTT, under the form of two recursors $\mathtt{rec}_+$ and $\mathtt{drec}_+$ translated in in Figure 4. To prove an extension of Proposition 2 to BTT, the main point is to check that $[\mathtt{drec}_+(t, X, u_1, u_2)]^{\mathtt{n}}$ as a type convertible to $[\sigma_{A+B}\ v\ X]^{\mathtt{n}}$. But actually, the return type of the recursor in the translation of $\mathtt{drec}_+$ has precisely been made to be of the form σ_{A+B}, convertible to the expected one up-to the use of the reduction rule of `let` and `force`.

C. Extending BTT *with linearity.*

There is a more direct translation of $\mathtt{drec}_+(v, X, u_1, u_2)$ provided that the translation of X is linear. Indeed, one can simply define

$$\begin{array}{l}
[\mathtt{drec}_+(t, X, u_1, u_2)]^{\mathtt{n}} := \mathtt{dlet}\ v := [t]^{\mathtt{n}}\ \mathtt{in} \\
\qquad \mathtt{rec}_+(v, \lambda v.\, [X]^{\mathtt{n}}\ (\mathtt{thunk}\ (\mathtt{return}\ v)), [u_1]^{\mathtt{n}}, [u_2]^{\mathtt{n}})
\end{array}$$

In that case, the translation of $\mathtt{drec}_+(t, X, u_1, u_2)$ has type $\mathtt{let}\ v := [t]^{\mathtt{n}}\ \mathtt{in}\ [X]^{\mathtt{n}}\ (\mathtt{thunk}\ (\mathtt{return}\ v))$ which is convertible to $[X]^{\mathtt{n}}\ (\mathtt{thunk}\ [t]^{\mathtt{n}})$ by linearity.

However, linearity of a predicate is a semantic notion and is undecidable in general. Ahman [17] provides a syntactic under-approximation of linearity and introduces the linear arrow $X \multimap Y$, which is intuitively the subtype of $\mathcal{U}\ X \to Y$ restricted to linear functions. This syntactic characterization captures in particular storage operators, but is slightly more general as it also allows the use of commutative cuts.

Defining a version of BTT where a similar syntactic restriction of linearity is used to generalize storage operators is beyond the scope of this paper.

V. Call-by-Value Translation

In this section, we describe the extension to a dependent setting of the call-by-value translation of simply typed λ-calculus. In particular, we show that this translation does not scale well to dependency indicating that call-by-value is not appropriate to deal with dependency.

A. Call-by-value translation in CBPV.

The standard by-value translation interprets types A as value types $[\![A]\!]^{\mathtt{v}}$ (in particular $[\![\Box_i]\!]^{\mathtt{v}} \equiv \Box_i^v$), and terms $t : A$ as computations of $[t]^{\mathtt{v}} : \mathcal{F}\ [\![A]\!]^{\mathtt{v}}$. But from $A : \Box_i$, we only get $[A]^{\mathtt{v}} : \mathcal{F}\ \Box_i^v$, and there is no way to define $[\![A]\!]^{\mathtt{v}} : \Box_i^v$ as using a `let` binding can only produce a computation type and not a value type (we come back to this problem in Section V-B). To solve this, we need to define $[\![A]\!]^{\mathtt{v}}$ primitively to $[A]^{\mathtt{v}}$, which is only possible if we know that A is a syntactic value. Therefore, the type theory CIC^v we can interpret must satisfy the following proposition.

Proposition 4. *If* $\Gamma \vdash_{\mathsf{CIC}^v} A : \Box$ *then* A *is a syntactic value.*

$$
\begin{array}{lcl}
[\mathtt{rec}_+(t, X, u_1, u_2)]^{\mathtt{n}} & := & \mathtt{let}\ v := [t]^{\mathtt{n}}\ \mathtt{in}\ \mathtt{rec}_+(v, \lambda\_.\, [X]^{\mathtt{n}}, [u_1]^{\mathtt{n}}, [u_2]^{\mathtt{n}}) \\
[\mathtt{drec}_+(t, X, u_1, u_2)]^{\mathtt{n}} & := & \mathtt{dlet}\ v := [t]^{\mathtt{n}}\ \mathtt{in}\ \mathtt{rec}_+(v, \lambda v.\, [\sigma_{A+B}]^{\mathtt{n}}\ (\mathtt{thunk}\ (\mathtt{return}\ v))\ (\mathtt{thunk}\ [X]^{\mathtt{n}}), [u_1]^{\mathtt{n}}, [u_2]^{\mathtt{n}})
\end{array}
$$

Fig. 4. CBN translation of recursors for coproducts

This can be obtained by stratifying the syntax into values and computations to enforce that types are always values (again, we only deal with coproducts, the other inductive types are translated in the same way).

$$
\begin{array}{ll}
\text{values} & A, B, v, w ::= \square_i \mid \Pi x : A.\, B \mid A + B \mid \\
 & \qquad \lambda x : A.\, t \mid x \mid \mathtt{inl}\ v \mid \mathtt{inr}\ v \\
\text{computations} & t, u ::= v \mid t\ u \mid \mathtt{let}\ x : A := t\ \mathtt{in}\ u \mid \\
 & \qquad \mathtt{rec}_+(v, A, u_1, u_2)
\end{array}
$$

We do not detail the typing rules of CIC^v as we do not want to dwell too much on it. Figure 5 presents the rule for application and `let` binding. The by-value translation can then be extended to a dependent setting by translating a value v as $[\![v]\!]^{\mathtt{v}}$ and a term t as $[t]^{\mathtt{v}}$.

Definition 9 (By-value translation)**.** The by-value translation is defined as follows

$$
\begin{array}{lcl}
[\![\square_i]\!]^{\mathtt{v}} & := & \square_i^v \\
[\![\Pi x : A.\, B]\!]^{\mathtt{v}} & := & \mathcal{U}\ (\Pi x : [\![A]\!]^{\mathtt{v}}.\, \mathcal{F}\ [\![B]\!]^{\mathtt{v}}) \\
[\![A + B]\!]^{\mathtt{v}} & := & [\![A]\!]^{\mathtt{v}} + [\![B]\!]^{\mathtt{v}} \\
[\![\lambda x : A.\, t]\!]^{\mathtt{v}} & := & \mathtt{thunk}\ (\lambda x : [A]^{\mathtt{v}}.\, [t]^{\mathtt{v}}) \\
[\![x]\!]^{\mathtt{v}} & := & x \\
[\![\mathtt{inl}\ v]\!]^{\mathtt{v}} & := & \mathtt{inl}\ [\![v]\!]^{\mathtt{v}} \\
[\![\mathtt{inr}\ v]\!]^{\mathtt{v}} & := & \mathtt{inr}\ [\![t]\!]^{\mathtt{v}} \\
[v]^{\mathtt{v}} & := & \mathtt{return}\ [\![v]\!]^{\mathtt{v}} \\
[t\ u]^{\mathtt{v}} & := & \mathtt{let}\ f := [t]^{\mathtt{v}}\ \mathtt{in} \\
 & & \mathtt{let}\ x := [u]^{\mathtt{v}}\ \mathtt{in}\ \mathtt{force}\ f\ x \\
[\mathtt{let}\ v := t\ \mathtt{in}\ u]^{\mathtt{v}} & := & \mathtt{let}\ x := [t]^{\mathtt{v}}\ \mathtt{in}\ [u]^{\mathtt{v}} \\
[\mathtt{rec}_+(v, A, u_1, u_2)]^{\mathtt{v}} & := & \mathtt{rec}_+([\![v]\!]^{\mathtt{v}}, A, [u_1]^{\mathtt{v}}, [u_2]^{\mathtt{v}})
\end{array}
$$

This translation satisfies a correctness property, distinguishing between values and computations.

Proposition 5. *If* $\Gamma \vdash v : A$ *then* $[\![\Gamma]\!]^{\mathtt{v}} \vdash_v [\![v]\!]^{\mathtt{v}} : [\![A]\!]^{\mathtt{v}}$ *and if* $\Gamma \vdash t : A$ *then* $[\![\Gamma]\!]^{\mathtt{v}} \vdash_c [t]^{\mathtt{v}} : \mathcal{F}\ [\![A]\!]^{\mathtt{v}}$.

Proposition 6. *If* $t \to_{\mathtt{v}} u$ *then* $[t]^{\mathtt{v}} \equiv [u]^{\mathtt{v}}$.

B. Limitation of call-by-value.

The theory induces by this stratification between values and computations is a very weak one. First, types are by construction restricted to be values. This means in particular that the `let` binder can not be dependent because otherwise its type would be a computation—similarly to the rule `dlet`/`let` of ∂CBPV. Secondly, there is no way to perform any kind of large elimination. Indeed, there are only two ways to use the variable introduced by a dependent product: either as a type itself when it lives in a universe, or by passing this variable to an indexed type (such as equality).

This theory could be extended to reflect more faithfully the ∂CBPV target, at a cost of a much more intricate syntax, i.e. by allowing chains of `let` bindings to the right hand side of a colon, and duplicating the application rule to reflect the `let`/`dlet` split. Such a theory would be way too complex to be explained shortly here, so we will refrain from doing it.

C. Recovering CIC through thunkability.

The stratification required by the call-by-value translation is necessary to syntactically know that

$$[A]^{\mathtt{v}} \equiv \mathtt{return}\ [\![A]\!]^{\mathtt{v}}$$

when A is a type. But there is another more semantic way of having a similar property, by ensuring that the translation of a term is always thunkable. This way, we know that the translation $[A]^{\mathtt{v}}$ of a type A is always effect-free and thus morally equivalent to $\mathtt{return}\ A'$ for some value type A'.

We make this intuition formal in the next section by defining a third translation into ∂CBPV.

VI. Call-by-Thunkable Translation

Building on the notion of effect compatibility, we describe in this section an embedding of all of CIC—including full dependent elimination and substitution—into an extension of ∂CBPV. The basic idea under this translation is twofold:

- First, we embed a call-by-value language in call-by-name, as we would do through a CPS. This solves the issue of types being restricted to values from Section V.
- Second, we restrict the computations to be observationally pure, by requiring them to be thunkable.

This technique is similar to Girard's boring translation [21] in linear logic. As it will turn out in Section IX, this is essentially what happens in the standard presheaf construction.

Definition 10. We extend ∂CBPV with a built-in notion of thunkability defined in Figure 6.

Intuitively, $\Theta\ A$ is the subset type of elements of $\mathcal{F}\ A$ that are thunkable. Values are in particular thunkable through θ, and one can project out the underlying element of $\mathcal{F}\ A$ with $\Downarrow$. The first equation axiomatizes thunkability, and the second the subset type constructor. Note that *per se*, the rules enforce that inhabitants of $\Theta\ A$ are necessarily values, but they are compatible with extensions introducing thunkable non-values.

A. Negative Fragment

Definition 11 (Thunkable translation)**.** The thunkable translation $[-]^{\theta}$ from CC_ω into ∂CBPV is defined as follows.

$$\frac{\Gamma \vdash t : \Pi x : A.\, B \qquad \Gamma \vdash v : A}{\Gamma \vdash t\ v : B\{x := v\}} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : \square_i \qquad \Gamma, x : A \vdash u : B}{\Gamma \vdash \mathtt{let}\ x : A := t\ \mathtt{in}\ u : B}$$

Fig. 5. Call-by-value type theory

$$X, Y, t, u ::= \ldots \mid \Theta\ A \mid \theta\ v \mid \Downarrow t$$

$$\frac{\Gamma \vdash A : \square_i^v}{\Gamma \vdash \Theta\ A : \square_i^c} \qquad \frac{\Gamma \vdash v : A}{\Gamma \vdash \theta\ v : \Theta\ A} \qquad \frac{\Gamma \vdash t : \Theta\ A}{\Gamma \vdash \Downarrow t : \mathcal{F}\ A}$$

$$\mathtt{let}\ x : A := \Downarrow t\ \mathtt{in}\ f\ (\mathtt{thunk}\ (\theta\ x)) \equiv f\ (\mathtt{thunk}\ t)$$
$$\Downarrow (\theta\ v) \equiv \mathtt{return}\ v$$

Fig. 6. Thunkable types

$$\begin{array}{lcl}
[\![A]\!]_c^\theta & := & \mathtt{let}\ A : \square_i^v := \Downarrow [A]^\theta\ \mathtt{in}\ \Theta\ A \\
[\![A]\!]^\theta & := & \mathcal{U}\ [\![A]\!]_c^\theta \\
[\square_i]^\theta & := & \theta\ \square_i^v \\
[\Pi x : A.\, B]^\theta & := & \theta\ (\mathcal{U}\ (\Pi x : [\![A]\!]^\theta.\, \Theta\ [\![B]\!]^\theta)) \\
[x]^\theta & := & \mathtt{force}\ x \\
[t\ u]^\theta & := & \mathtt{let}\ f := \Downarrow [t]^\theta\ \mathtt{in} \\
 & & \mathtt{let}\ r := \Downarrow (\mathtt{force}\ f\ (\mathtt{thunk}\ [u]^\theta))\ \mathtt{in} \\
 & & \mathtt{force}\ r \\
[\lambda x : A.\, t]^\theta & := & \theta\ (\mathtt{thunk}\ (\lambda x : [\![A]\!]^\theta.\, \theta\ (\mathtt{thunk}\ [t]^\theta)))
\end{array}$$

As explained above, it is fairly obvious that this translation mixes call-by-name features (e.g. arguments are thunked) with call-by-value ones (e.g. the translation of Π and $\square$).

Proposition 7. *We have for any terms t and r $[t\{x := r\}]^\theta \equiv_{\partial\mathsf{CBPV}} [t]^\theta\{x := \mathtt{thunk}\ [r]^\theta\}$.*

Proposition 8. *If $t \equiv_{\mathsf{CC}_\omega} u$ then $[t]^\theta \equiv_{\partial\mathsf{CBPV}} [u]^\theta$.*

Proposition 9. *If $\Gamma \vdash_{\mathsf{CC}_\omega} t : A$ then $[\![\Gamma]\!]^\theta \vdash_c [t]^\theta : [\![A]\!]_c^\theta$.*

B. Positive Fragment

Again, we only present the translation for coproduct types, as it is similar for other inductive types.

Definition 12. The thunkable translation of coproduct types is defined below.

$$\begin{array}{lcl}
[A + B]^\theta & := & \theta\ ([\![A]\!]^\theta + [\![B]\!]^\theta) \\
[\mathtt{inl}\ t]^\theta & := & \theta\ (\mathtt{inl}\ (\mathtt{thunk}\ [t]^\theta)) \\
[\mathtt{inr}\ u]^\theta & := & \theta\ (\mathtt{inr}\ (\mathtt{thunk}\ [u]^\theta)) \\
[\mathtt{rec}_+(t, P, u_1, u_2)]^\theta & := & \mathtt{dlet}\ s := \Downarrow [t]^\theta\ \mathtt{in} \\
 & & \mathtt{rec}_+(s, \tilde{P}, \tilde{u}_1, \tilde{u}_2)
\end{array}$$

where

- $\tilde{P} := \lambda s.\, [\![P\ s_0]\!]_c^\theta\{s_0 := \mathtt{thunk}\ (\theta\ s)\}$
- $\tilde{u}_i := \lambda x.\, \mathtt{let}\ u_i := \Downarrow [u_i]^\theta\ \mathtt{in}$
  $\mathtt{let}\ r := \Downarrow (\mathtt{force}\ u_i\ x)\ \mathtt{in}\ \mathtt{force}\ r$

Proposition 10. *Assuming ∂CBPV enjoys the η-rule below,*

$$\mathtt{thunk}\ (\mathtt{force}\ t) \equiv t$$

$[-]^\theta$ interprets coproducts with full dependent elimination.

Proof. The difficult case consists in proving that the eliminator has the expected CIC type, the other ones being straightforward. Two things need to be proved thus:

- first that the eliminator is well-typed, i.e.
$$\begin{array}{rcl} \tilde{u}_1 & : & \Pi x : [\![A]\!]^\theta.\, \tilde{P}\ (\mathtt{inl}\ x) \\ \tilde{u}_2 & : & \Pi y : [\![B]\!]^\theta.\, \tilde{P}\ (\mathtt{inr}\ y) \end{array}$$
- second that its type is convertible to the one of full dependent elimination, i.e.
$$\mathtt{let}\ s := \Downarrow [s]^\theta\ \mathtt{in}\ \tilde{P}\ s \equiv [\![P\ s]\!]_c^\theta$$

The first part is immediate, but as a technicality requires the above η-rule, which holds definitionally in all the ∂CBPV models of this paper. The second part is a direct but crucial application of the thunkability of $[s]^\theta$ and Proposition 7. □

This translation can be extended to any inductive type, as long as there is a ∂CBPV countepart. Therefore:

Theorem 2. *The thunkable translation is a model of* CIC.

This shows that to interpret all of CIC into ∂CBPV, it is necessary to be explicit about the absence of effects in the term, using the notion of thunkability. This way, as all terms are thunkable, all predicates are linear, and thus we get both substitution and large dependent elimination.

VII. Reader Translation

The reader translation is a very simple model of ∂CBPV that corresponds computationally to the addition of a global cell. This cell can be read, hence the name, and can also be updated in a well-scoped way, i.e. the update cannot escape from the term being evaluated.

Definition 13 (Reader translation)**.** We assume a type for the cell $\vdash_{\mathsf{CIC}} \mathbb{P} : \square_0$, and define the reader translation from ∂CBPV into CIC in Figure 7.

Note how the translation of computations systematically starts with an abstraction over $p : \mathbb{P}$, the current global cell.

Theorem 3 (Soundness)**.** *The following hold.*

- *$\Gamma \vdash_c t : X$ implies $[\![\Gamma]\!] \vdash [t] : [\![X]\!]^c$,*
- *$\Gamma \vdash_v v : A$ implies $[\![\Gamma]\!] \vdash [v] : [\![A]\!]^v$*
- *$t \equiv u$ implies $[t] \equiv [u]$ and similarly for values.*

We show how to extend this model to ∂CBPV coproduct type. Other inductive types are treated similarly.

Definition 14. We translate coproducts as follows.

$$\begin{array}{lcl}
[A + B] & := & [\![A]\!]^v + [\![B]\!]^v \\
[\mathtt{inl}\ v] & := & \mathtt{inl}\ [v] \\
[\mathtt{inr}\ w] & := & \mathtt{inr}\ [w] \\
[\mathtt{rec}_+(v, X, u_1, u_2)] & := & \lambda p : \mathbb{P}.\, \mathtt{rec}_+([v], \tilde{X}, \tilde{u}_1, \tilde{u}_2)
\end{array}$$

$$
\begin{array}{lcl}
[\![X]\!]^c & := & \Pi p : \mathbb{P}.\, [X]\ p \\
[\![A]\!]^v & := & [A] \\
[\square_i^c] & := & \lambda p : \mathbb{P}.\, \square_i \\
[\square_i^v] & := & \square_i \\
[\mathcal{U}\ X] & := & \Pi p : \mathbb{P}.\, [X]\ p \\
[\mathcal{F}\ A] & := & \lambda p : \mathbb{P}.\, [A] \\
[\Pi x : A.\, X] & := & \lambda p : \mathbb{P}.\, \Pi x : [\![A]\!]^v.\, [X]\ p \\
[x] & := & x \\
[t\ v] & := & \lambda p : \mathbb{P}.\, [t]\ p\ [v] \\
[\lambda x : A.\, t] & := & \lambda p : \mathbb{P}.\, \lambda x : [\![A]\!]^v.\, [t]\ p \\
[\mathtt{thunk}\ t] & := & [t] \\
[\mathtt{force}\ v] & := & \lambda p : \mathbb{P}.\, [v]\ p \\
[\mathtt{return}\ v] & := & \lambda p : \mathbb{P}.\, [v] \\
[\mathtt{let}\ x : A := t\ \mathtt{in}\ u] & := & \lambda p : \mathbb{P}.\, (\lambda x : [\![A]\!]^v.\, [u]\ p)\ ([t]\ p) \\
[\mathtt{dlet}\ x : A := t\ \mathtt{in}\ u] & := & \lambda p : \mathbb{P}.\, (\lambda x : [\![A]\!]^v.\, [u]\ p)\ ([t]\ p)
\end{array}
$$

Fig. 7. Reader Translation

where

- $\tilde{X} := \lambda s.\, [X]\ p\ s$
- $\tilde{u}_1 := \lambda x.\, [u_1]\ p\ x$
- $\tilde{u}_2 := \lambda y.\, [u_2]\ p\ y$

Soudness is easily showed to be preserved by this extension.

Theorem 4. *The reader translation is a model of* ∂CBPV.

This model is simple enough to understand clearly what goes wrong in call-by-name dependent elimination, and why the restriction to BTT is necessary.

Example 2. Unfolding the translations, implementing dependent elimination for coproducts in $[[-]^{\mathtt{n}}]$ amounts to, given

- $A, B : \mathbb{P} \to \square$
- $s : \mathbb{P} \to [\![A]\!]^c + [\![B]\!]^c$
- $P : \mathbb{P} \to (\mathbb{P} \to [\![A]\!]^c + [\![B]\!]^c) \to \square$
- $u_1 : \Pi p : \mathbb{P}.\, \Pi x : [\![A]\!]^c.\, P\ p\ (\lambda q : \mathbb{P}.\, \mathtt{inl}\, x)$
- $u_2 : \Pi p : \mathbb{P}.\, \Pi y : [\![B]\!]^c.\, P\ p\ (\lambda q : \mathbb{P}.\, \mathtt{inr}\, y)$

being able to produce, barring conversion constraints, some

$$e : \Pi p : \mathbb{P}.\, P\ p\ s$$

But there is little hope! If $\mathbb{P}$ is not a singleton, there is no reason for s to be either constantly left or constantly right. For instance, if $\mathbb{P} := \mathbb{B}$, there are extensionally four possible cases for the shape of s, but the hypotheses only cover the two constant ones. Thus dependent elimination fails.

VIII. Weaning Translation in ∂CBPV

A. Weaning translation.

This section describes the weaning translation from ∂CBPV into CIC, based on the notion of *self-algebraic proto-monad*. See [11] for a more in-depth description of this structure.

Definition 15 ([11])**.** A self-algebraic proto-monad is given by the following universe-polymorphic family of CIC terms:

- $\mathtt{T} : \square_i \to \square_i$
- $\mathtt{ret} : \Pi A : \square_i.\, A \to \mathtt{T}\ A$
- $\mathtt{bnd} : \Pi (A : \square_i)\, (B : \square_j).\, \mathtt{T}\ A \to (A \to \mathtt{T}\ B) \to \mathtt{T}\ B$
- $\mathtt{El} : \mathtt{T}\ \square\!\square_i \to \square\!\square_i$
- $\mathtt{hbnd} : \Pi (A : \square_i)\, (B : \mathtt{T}\ \square\!\square_j).$
  $\mathtt{T}\ A \to (A \to (\mathtt{El}\ B).\pi_1) \to (\mathtt{El}\ B).\pi_1$

where

$$\square\!\square_i := \Sigma A : \square_i.\, \mathtt{T}\ A \to A$$

furthermore subject to the following definitional equations:

$$
\begin{array}{lcl}
\mathtt{El}\ (\mathtt{ret}\ \square\!\square_i\ A) & \equiv & A \\
\mathtt{bnd}\ A\ B\ (\mathtt{ret}\ A\ t)\ f & \equiv & f\ t \\
\mathtt{hbnd}\ A\ B\ (\mathtt{ret}\ A\ t)\ f & \equiv & f\ t.
\end{array}
$$

For brevity, we will use implicit type arguments for the monadic combinators. Similarly, when forming inhabitants of $\square\!\square$ in the translation below, we will omit the algebra morphism and simply write $\{\!\{A\}\!\} : \square\!\square$ given any $A : \square$. The morphisms are the same as in our previous paper [11] and are canonical, e.g. for a type of the form $\mathtt{T}\ A$, it is the free morphism.

Definition 16 (Weaning)**.** Assuming a self-algebraic proto-monad, we define the weaning translation from ∂CBPV to CIC by induction over the term syntax in Figure 8. We only deal with coproducts, dependent sums and equality can be handled in the same way.

This translation gives rise to a syntactical model of ∂CBPV.

Proposition 11 (Soundness)**.** *The following hold.*

- $\Gamma \vdash_c t : X$ *implies* $[\![\Gamma]\!] \vdash_{\mathsf{CIC}} [t] : [\![X]\!]^c$.
- $\Gamma \vdash_v v : A$ *implies* $[\![\Gamma]\!] \vdash_{\mathsf{CIC}} [t] : [\![A]\!]^v$.
- $t \equiv_{\partial\mathsf{CBPV}} u$ *implies* $[t] \equiv_{\mathsf{CIC}} [u]$ *and similarly for values.*

Proof. Similar to [11]. □

Corollary 1. *The weaning translation is a model of* ∂CBPV.

IX. Forcing Translation in ∂CBPV

This section is devoted to the definition of the forcing model of ∂CBPV. Contrarily to the other ∂CBPV models presented in this paper, we require the target theory to be extensional, in the sense that propositional and definitional equality must coincide. Such a theory will be denoted as ECIC. This requirement is slightly stronger than [19] to be able to interpret ∂CBPV conversion by conversion in the target.

A. Forcing Translation

First of all, we need a notion of base category in the target.

Definition 17 (Base category)**.** A base category is given by the four ECIC terms below:

- $\mathbb{P} : \square_0$;
- $\leq : \mathbb{P} \to \mathbb{P} \to \square_0$;
- $\mathbf{1} : \Pi p : \mathbb{P}.\, p \leq p$;
- $* : \Pi (p\, q\, r : \mathbb{P}).\, p \leq q \to q \leq r \to p \leq r$

subject to the following conversion rules.

$$
\begin{array}{lcl}
[\![X]\!]^c & := & (\texttt{El}\ [X]).\pi_1 \\
[\![A]\!]^v & := & [A] \\
[\square_i^c] & := & \texttt{ret}\ \{\!\{\texttt{T}\ \square_i\}\!\} \\
[\square_i^v] & := & \square_i \\
[\mathcal{F}\ A] & := & \texttt{ret}\ \{\!\{\texttt{T}\ A\}\!\} \\
[\mathcal{U}\ X] & := & [\![X]\!]^c \\
[x] & := & x \\
[\texttt{thunk}\ t] & := & [t] \\
[\texttt{force}\ v] & := & [v] \\
[\lambda x : A.\, t] & := & \lambda x : [\![A]\!]^v.\, [t] \\
[t\ v] & := & [t]\ [v] \\
[\Pi x : A.\, X] & := & \texttt{ret}\ \{\!\{\Pi x : [\![A]\!]^v.\, [\![X]\!]^c\}\!\} \\
[\texttt{let}\ x : A := t\ \texttt{in}\ u] & := & \texttt{bnd}\ [t]\ (\lambda x : [\![A]\!]^v.\, [u]) \\
[\texttt{dlet}\ x : A := t\ \texttt{in}\ u] & := & \texttt{hbnd}\ [t]\ (\lambda x : [\![A]\!]^v.\, [u]) \\
[\texttt{return}\ v] & := & \texttt{ret}\ [v] \\
[A + B] & := & [\![A]\!]^v + [\![B]\!]^v \\
[\texttt{inl}\ v] & := & \texttt{inl}\ [v] \\
[\texttt{inr}\ w] & := & \texttt{inr}\ [w] \\
[\texttt{rec}_+(v, X, u_1, u_2)] & := & \texttt{rec}_+([v], [X], [u_1], [u_2]) \\
[\![\cdot]\!] & := & \cdot \\
[\![\Gamma, x : A]\!] & := & [\![\Gamma]\!], x : [\![A]\!]^v
\end{array}
$$

Fig. 8. Weaning Translation

$$
\begin{array}{lcl}
[\![A]\!]^\Gamma_p & := & [A]^\Gamma_p.\texttt{type}\ p\ \mathbf{1}_p \\
[\square_i^c]^\Gamma_p & := & \square_i \\
[\square_i^v]^\Gamma_p & := & \{\!\{\lambda(q\,\alpha : p).\, \square_i\ q\}\!\} \\
[x]^\Gamma_p & := & x \\
[\texttt{thunk}\ t]^\Gamma_p & := & \lambda(q\,\alpha : p).\, \alpha \bullet_\Gamma [t]^\Gamma_q \\
[\texttt{force}\ v]^\Gamma_p & := & [v]^\Gamma_p\ p\ \mathbf{1}_p \\
[\lambda x : A.\, t]^\Gamma_p & := & \lambda x : [\![A]\!]^\Gamma_p.\, [t]^{\Gamma, x:A}_p \\
[t\ v]^\Gamma_p & := & [t]^\Gamma_p\ [v]^\Gamma_p \\
[\Pi x : A.\, X]^\Gamma_p & := & \Pi x : [\![A]\!]^\Gamma_p.\, [X]^\Gamma_p \\
[\texttt{let}\ x : A := t\ \texttt{in}\ u]^\Gamma_p & := & (\lambda x : [\![A]\!]_p.\, [u]^{\Gamma, x:A}_p)\ [t]^\Gamma_p \\
[\texttt{dlet}\ x : A := t\ \texttt{in}\ u]^\Gamma_p & := & (\lambda x : [\![A]\!]_p.\, [u]^{\Gamma, x:A}_p)\ [t]^\Gamma_p \\
[\texttt{return}\ v]^\Gamma_p & := & [v]^\Gamma_p \\
[\mathcal{U}\ X]^\Gamma_p & := & \{\!\{\lambda(q\,\alpha : p).\, \Pi(r\,\beta : q). \\
 & & \qquad (\beta * \alpha) \bullet_\Gamma [X]^\Gamma_r\}\!\} \\
[\mathcal{F}\ A]^\Gamma_p & := & [\![A]\!]^\Gamma_p \\
[\![\cdot]\!]_p & := & p : \mathbb{P} \\
[\![\Gamma, x : A]\!]_p & := & [\![\Gamma]\!]_p, x : [\![A]\!]^\Gamma_p
\end{array}
$$

Assuming $\alpha : q \leq p$, we write

$$
\begin{array}{lcl}
\alpha \circ^\Gamma_A t & := & [A]^\Gamma_p.\texttt{mono}\ p\ \mathbf{1}_p\ q\ \alpha\ t \\
\alpha \bullet_\Gamma t & := & t\{x_i := \alpha \circ^\Gamma_{A_i} x_i \mid (x_i : A_i) \in \Gamma\}.
\end{array}
$$

Fig. 9. Forcing Translation

$$\mathbf{1}_p * f \equiv f \quad f * \mathbf{1}_q \equiv f \quad f * (g * h) \equiv (f * g) * h$$

For clarity, we use an infix notation with implicit type arguments. While we stick to a notation reminiscent of preorders, the relation is not necessarily proof-irrelevant. We assume in the remainder of this section a fixed base category that we will call *forcing conditions*. We will use the binder notation $(q\,\alpha : p)$ for $(q : \mathbb{P})\,(\alpha : q \leq p)$.

Definition 18 (Monotonic types)**.** Given $p : \mathbb{P}$ and a universe index i, we define $\square_i\ p$, the type of monotonic types at p as:

$$
\left\{
\begin{array}{l}
\texttt{type} : \Pi(q\,\alpha : p).\, \square_i \\
\texttt{mono} : \Pi(q\,\alpha : p)\,(r\,\beta : q). \\
\qquad \texttt{type}\ q\ \alpha \to \texttt{type}\ r\ (\beta * \alpha) \\
\texttt{refl} : \Pi(q\,\alpha : p)\,(x : \texttt{type}\ q\ \alpha). \\
\qquad \texttt{mono}\ q\ \alpha\ q\ \mathbf{1}_q\ x = x \\
\texttt{trns} : \Pi(q\,\alpha : p)\,(r\,\beta : q)\,(s\,\gamma : r)\,(x : \texttt{type}\ q\ \alpha). \\
\qquad \texttt{mono}\ q\ \alpha\ s\ (\gamma * \beta)\ x = \\
\qquad\quad \texttt{mono}\ r\ (\beta * \alpha)\ s\ \gamma\ (\texttt{mono}\ q\ \alpha\ r\ \beta\ x)
\end{array}
\right\}
$$

We use a record notation for readability, but this can readily be understood as an iterated Σ-type. In what follows, given $A : \Pi(q\,\alpha : p).\, \square_i$ we will just write $\{\!\{A\}\!\}$ for the monotonic type with field $\texttt{type} : A$ if the other fields are canonical.

Definition 19 (Forcing translation)**.** The forcing translation, defined at Fig. 9, is indexed by a ∂CBPV environment Γ and a condition p and produces an ECIC term.

Let us gloss over this definition. Computation types are interpreted as mere types, while value types are interpreted as *monotonic* types. In particular as $\square_i^v : \square_{i+1}^v$, one needs to equip the value universe with that structure as well. As mentioned above, we omit non-`type` fields.

The $\mathcal{F}$ former simply forgets about the additional structure, while $\mathcal{U}$ freely turns a $\mathbb{P}$-indexed type into a monotonic one by quantifying over all lower conditions. Every time we quantify over a forcing condition $(q\,\alpha : p)$, we need to lift all the free variables of the subterms by making α act on them. This happens in the `thunk` and $\mathcal{U}$ cases. We need to do this because there is a mismatch between free variables which live at level q while we would like them to live at level p. Dually, the `force` translation *resets* a boxed term by applying it to the current condition.

Remark 2. While at first sight the reader translation looks like a trivialization of the forcing translation where the base category is a full preorder, it is not the case. The monotonicity requirement is absent from the reader translation, and consequently the lifting of open variables is also non-existent there.

Proposition 12. *If* $[\![\Gamma]\!]_p \vdash_{\mathsf{ECIC}} t : A$*, then*

- $\mathbf{1}_p \bullet_\Gamma t \equiv t$
- $(\beta * \alpha) \bullet_\Gamma t \equiv \alpha \bullet_\Gamma (\beta \bullet_\Gamma t)$

assuming well-typedness of morphisms.

Proposition 13 (Typing soundness)**.** *Assume* $\Gamma \vdash_c t : X$*, then* $[\![\Gamma]\!]_p \vdash [t]^\Gamma_p : [X]^\Gamma_p$ *and similarly for values.*

Proposition 14 (Computational soundness)**.** *For all* $\Gamma \vdash_c t,u : A$*, if* $t \equiv u$ *then* $[t]_p^\Gamma \equiv [u]_p^\Gamma$ *and similarly for values.*

As usual, those properties need to be proved by mutual induction. Extensionality in the target is critical for them to hold though, because we need to rewrite equations coming from `refl` and `trns` in arbitrary contexts.

As for presheaves, inductive types are translated pointwise. Once again, we only describe the coproduct translation.

Definition 20. Coproducts are translated as follows.

$$
\begin{aligned}
[A + B]_p^\Gamma &:= \{\!\{\lambda(q\,\alpha : p).\,(\alpha \bullet_\Gamma [\![A]\!]_q^\Gamma) + (\alpha \bullet_\Gamma [\![B]\!]_q^\Gamma)\}\!\} \\
[\mathtt{inl}\, v]_p^\Gamma &:= \mathtt{inl}\, [v]_p^\Gamma \\
[\mathtt{inr}\, w]_p^\Gamma &:= \mathtt{inr}\, [w]_p^\Gamma \\
[\mathtt{rec}_+(v, X, u_1, u_2)]_p^\Gamma &:= \mathtt{rec}_+([v]_p^\Gamma, [X]_p^\Gamma, [u_1]_p^\Gamma, [u_2]_p^\Gamma)
\end{aligned}
$$

Soundness is trivially extended.

Theorem 5. *The forcing translation is a model of* ∂CBPV.

This model is quite interesting. It is *not* the usual presheaf construction, because we do not require naturality of functions. As such, precomposing it with the by-name or by-value translation into ∂CBPV would not provide a model of CIC, but rather of BTT or CIC^v, forfeiting respectively dependent elimination or arbitrary substitution. Note that precomposing with the by-name translation essentially gives the forcing translation defined in [10]. But what would the thunkable translation look like? To answer this, we need to peek at the forcing translation of thunkability. Without further ado:

Proposition 15. *A term* $\Gamma \vdash_{\partial\mathsf{CBPV}} t : \mathcal{F}\, A$ *is thunkable iff it is natural, i.e. for all* $p, q, \alpha : q \leq p$*,*

$$\alpha \bullet_\Gamma [t]_q^\Gamma \equiv_{\mathsf{ECIC}} \alpha \circ_A^\Gamma [t]_p^\Gamma.$$

This naturality property matches exactly the one from categorical presheaves, and can be rewritten abstractly as

$$
\begin{array}{ccc}
[\![\Gamma]\!]_p & \xrightarrow{[t]_p^\Gamma} & [\![A]\!]_p^\Gamma \\
{\scriptstyle \alpha \bullet_\Gamma (-)}\downarrow & & \downarrow{\scriptstyle \alpha \circ_A^\Gamma (-)} \\
[\![\Gamma]\!]_q & \xrightarrow[[t]_q^\Gamma]{} & [\![A]\!]_q^\Gamma
\end{array}
$$

To the best of our knowledge, this property had never been observed in the literature so far, even though it is far-reaching. This is the deep reason why usual presheaves actually provide a full model of dependent type theory, including both substitution and dependent elimination. Namely, the presheaf category is at its very core the restriction of an effectful type theory to thunkable terms. The only place where this needs to be performed is on non-value types, that is, Π-types.

The trade-off of this construction is that thunkability is a quite extensional property, which does not matter in topos theory, but is not easily amenable in type theory. In particular, it remains an open problem to describe a syntactic presheaf model targetting intensional CIC.

X. Related Work

With the growing popularity of dependent type theory, there has been a recent surge in interest around this topic. In addition to the authors' papers [10], [11], [13], there are two lines of work that are particularly close to the current one, namely Ahman's [17] and Vákár's [16], [22].

We concede that Ahman's eMLTT system is very similar in spirit to our solution, as it features a restriction based on algebra homomorphisms, which are the categorical equivalent of linear morphisms. Unfortunately, he falls short of providing a satisfying account of large elimination. The type structure of his system embeds no less than a copy of an effect-free CIC to handle type-level computation, which prevents decomposing large elimination through the usual CBPV route. The current paper solves this limitation straightforwardly by equipping computation types, seen as terms, with an algebra structure. Furthermore, categorical models are of little use when it comes to intensional properties, which is why we argue in favour of our syntactic models.

In [11], we already advocated against Vákár's dCBPV. For the sake of self-containedness, we summarize our grievances here. Vákár's system is twofold. First, he gives a base system that is very weak where there is no real way to depend on terms, similarly to the system we sketch in Section V. Upon realizing that the system is mostly useless, he adds a extension rule allowing to lift any inhabitant of free computations in types. Unluckily, this rule is equivalent to postulating that all predicates are linear, which, by the Fire Triangle, means that any model satisfying this rule is either pure or inconsistent.

Let us also mention an interesting recent attempt by Bowman et al. [23] at working around the negative result of Barthe and Uustalu regarding a type-preserving CPS of CIC. Their model fundamentally relies on an impredicative universe as well as an extensional target, which we find a bit disappointing. Nonetheless, the parametricity equation that they admit as a free theorem is no more than thunkability through the CPS translation, which makes their system a sibling of the by-thunkable decomposition.

We will end by citing a slightly more distant, but still related work by Lepigre [9]. Although his system is somewhere in between NuPRL and HOL, he faces the same issues of interactions between effects and dependency, this time in a call-by-value setting. In order for the system to be usable in practice, this naturally leads him to extend value restriction from a syntactic criterion to a semantic one akin to thunkability.

Acknowledgements

This research has been funded by the CoqHoTT ERC Grant 637339.

References

[1] V. Glivenko, "Sur quelques points de la logique de M. Brouwer," *Bulletins de la classe des sciences*, vol. 15, pp. 183–188, 1929.

[2] E. Moggi, "Notions of computation and monads," *Information and Computation*, vol. 93, no. 1, pp. 55–92, Jul. 1991.

[3] G. Barthe and T. Uustalu, "Cps translating inductive and coinductive types," in *Proceedings of Partial Evaluation and Semantics-based Program Manipulation*. ACM, 2002, pp. 131–142.

[4] H. Herbelin, "On the degeneracy of sigma-types in presence of computational classical logic," in *Seventh International Conference, TLCA '05, Nara, Japan. April 2005, Proceedings*, ser. Lecture Notes in Computer Science, P. Urzyczyn, Ed., vol. 3461. Springer, 2005, pp. 209–220.

[5] P. Martin-Löf, "100 years of zermelo's axiom of choice: what was the problem with it?" *Comput. J.*, vol. 49, no. 3, pp. 345–350, 2006. [Online]. Available: https://doi.org/10.1093/comjnl/bxh162

[6] P. B. Levy, "Call-by-push-value," Ph.D. dissertation, Queen Mary, University of London, 2001.

[7] T. Griffin, "A formulae-as-types notion of control," in *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*, 1990, pp. 47–58. [Online]. Available: https://doi.org/10.1145/96709.96714

[8] A. Wright, "Simple imperative polymorphism," in *LISP and Symbolic Computation*, 1995, pp. 343–356.

[9] R. Lepigre, "A classical realizability model for a semantical value restriction," in *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, 2016, pp. 476–502. [Online]. Available: https://doi.org/10.1007/978-3-662-49498-1_19

[10] G. Jaber, G. Lewertowski, P.-M. Pédrot, M. Sozeau, and N. Tabareau, "The definitional side of the forcing," in *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, 2016, pp. 367–376.

[11] P. Pédrot and N. Tabareau, "An effectful way to eliminate addiction to dependence," in *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. IEEE Computer Society, 2017, pp. 1–12. [Online]. Available: https://doi.org/10.1109/LICS.2017.8005113

[12] J.-L. Krivine, "Classical logic, storage operators and second-order lambda-calculus," *Ann. Pure Appl. Logic*, vol. 68, no. 1, pp. 53–78, 1994.

[13] P. Pédrot and N. Tabareau, "Failure is not an option - an exceptional type theory," in *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, 2018, pp. 245–271. [Online]. Available: https://doi.org/10.1007/978-3-319-89884-1_9

[14] G. Munch-Maccagnoni, "Models of a Non-Associative Composition," in *17th International Conference on Foundations of Software Science and Computation Structures*, A. Muscholl, Ed., vol. 8412. Grenoble, France: Springer, Apr. 2014, pp. 396–410.

[15] P. B. Levy, "Contextual isomorphisms," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. New York, NY, USA: ACM, 2017, pp. 400–414.

[16] M. Vákár, "A framework for dependent types and effects," 2015, draft. [Online]. Available: https://arxiv.org/abs/1512.08009

[17] D. Ahman, "Handling fibred algebraic effects," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 7:1–7:29, Jan. 2018. [Online]. Available: http://doi.acm.org/10.1145/3158095

[18] S. Boulier, P.-M. Pédrot, and N. Tabareau, "The next 700 syntactical models of type theory," in *Proceedings of Certified Programs and Proofs*. ACM, 2017, pp. 182–194.

[19] G. Jaber, N. Tabareau, and M. Sozeau, "Extending Type Theory with Forcing," in *LICS 2012 : Logic In Computer Science*, Dubrovnik, Croatia, Jun. 2012, pp. 0–0.

[20] C. Führmann, "Direct models of the computational lambda-calculus," *Electronic Notes in Theoretical Computer Science*, vol. 20, pp. 245 – 292, 1999, mFPS XV, Mathematical Foundations of Progamming Semantics, Fifteenth Conference. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066104800781

[21] J. Girard, "Linear logic," *Theor. Comput. Sci.*, vol. 50, pp. 1–102, 1987. [Online]. Available: https://doi.org/10.1016/0304-3975(87)90045-4

[22] M. Vákár, "In search of effectful dependent types," Ph.D. dissertation, University of Oxford, 2017.

[23] W. J. Bowman, Y. Cong, N. Rioux, and A. Ahmed, "Type-preserving CPS translation of Σ and Π types is not not possible," *PACMPL*, vol. 2, no. POPL, pp. 22:1–22:33, 2018. [Online]. Available: https://doi.org/10.1145/3158110