The Fire Triangle

Pierre-Marie Pédrot, Nicolas Tabareau

The Fire Triangle

How to Mix Substitution, Dependent Elimination, and Effects

PIERRE-MARIE PÉDROT, Inria, France
NICOLAS TABAREAU, Inria, France

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.

CCS Concepts: • **Theory of computation → Type theory**.

Additional Key Words and Phrases: Type Theory, Effects

ACM Reference Format:
Pierre-Marie Pédrot and Nicolas Tabareau. 2020. The Fire Triangle: How to Mix Substitution, Dependent Elimination, and Effects. *Proc. ACM Program. Lang.* 4, POPL, Article 58 (January 2020), 28 pages. https://doi.org/10.1145/3371126

1 INTRODUCTION

The addition of effects to a logical system via syntactic translations is not new and can be traced back to double-negation translations [Glivenko 1929], although the modern standpoint can undoubtedly be attributed to Moggi in his seminal paper [Moggi 1991].

Since the inception of dependent type theory, several people tried to apply the techniques coming from simply-typed settings to enrich it with new reasoning 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 [Barthe and Uustalu 2002], and similarly Herbelin proved that CIC was inconsistent with computational classical logic [Herbelin 2005].

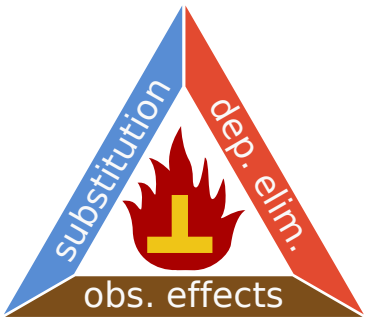


Fig. 1. The substitution-dependent elimination-observable effects triangle

Retrospectively, this should not have been that surprising. This incompatibility is the reflection of a very ancient issue: mixing of classical logic with the axiom of choice, whose intuitionistic version is a consequence of

Authors' addresses: Pierre-Marie Pédrot, Inria, Gallinette Project-Team, Nantes, France; Nicolas Tabareau, Inria, Gallinette Project-Team, Nantes, France.

2475-1421/2020/1-ART58
https://doi.org/10.1145/3371126

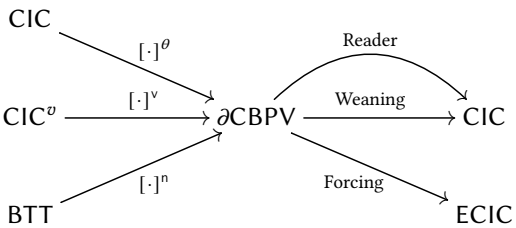
dependent elimination, is a well-known source of foundational problems [Martin-Löf 2006]. While in the literature much emphasis has been put on the particular case of classical logic, we argue in this paper that this is an instance of a broader phenomenon, namely that observable side-effects are at odds 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 2.

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 [Levy 2001] to the dependently-typed setting, coined ∂CBPV, whose design has been fueled by the recent work of the authors' on effectful type theories. In particular, we provide syntactic models of ∂CBPV to justify it, as well as decompositions into it arising from the usual embeddings into CBPV. This new theory explains for instance the recent attempt by Bowman et al. [Bowman et al. 2018] at working around the negative result of Barthe and Uustalu regarding a type-preserving CPS of CIC using a parametricity equation which corresponds exactly to the general notion of thunkability in ∂CBPV.

1.1 Plan of the Paper

In Section 2, 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 3 describes ∂CBPV, a system that allows to safely mix effects and dependent type theory. In Sections 5, 6 and 7, we give syntactic decompositions from several flavours of type theory into ∂CBPV. Dually, Sections 8, 9 and 10 provide syntactic models of ∂CBPV based on previous work. All of these translations are summarized bellow.



We then show how ∂CBPV can be instantiated with various effects, giving rise for instance to a generic `catch` operator, an elegant setting for normalization by evaluation, or a generalization of the open modality. Section 11 provides further comparisons with similar attempts.

2 SUBSTITUTION, DEPENDENT ELIMINATION AND OBSERVABLE EFFECTS

2.1 The Fire Triangle: A General No-Go Theorem

Let us now make precise this tension between substitution, dependent elimination and observable 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 two different typing judgments:

$$\Gamma \vdash t : A$$

saying that t has type A in context Γ and

$$\Gamma \vdash \star : A$$

saying that A is inhabited in context Γ.

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

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

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

To express what it means to feature dependent elimination and observable 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 $\vdash \mathtt{true} : \mathbb{B}$ and $\vdash \mathtt{false} : \mathbb{B}$. 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 inhabited when x is substituted by `true` and by `false`, then it is inhabited 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).

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

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 observably 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 observably effectful is to say that there exists a boolean term which is not observationally equivalent to `true` nor `false`.

Definition 3 (Observable Effects)**.** A type theory is *observably 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 $C[\mathtt{true}] \equiv \mathtt{true}$ and $C[\mathtt{false}] \equiv \mathtt{true}$, but $C[t] \equiv \mathtt{false}$ (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 observable 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 observably 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 3. By dependent elimination, it holds that $x : \mathbb{B} \vdash \star : C[x] = \mathtt{true}$. By substitution, $\vdash \star : C[t] = \mathtt{true}$. By conversion and because $C[t] \equiv \mathtt{false}$, this implies $\vdash \star : \mathtt{false} = \mathtt{true}$.

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

Example 1. An archetypical example of an observably effectful term can be obtained with `callcc` [Griffin 1990]. It is indeed possible to use it to write a term $\mathtt{decide} : \square \to \mathbb{B}$ that decides whether a type is inhabited. Obviously, $\mathtt{decide}\ A$ cannot enjoy canonicity in general. Such booleans are called *backtracking* or *non-standard*, and are the root of Herbelin's paradox.

Non Example 1. The definition of observable effects does not capture all effects in the literature. For instance, it does not apply to a printing operation, the presence of exception without handlers, or non-termination. In those cases, it is not possible to reason on effects in the type theory.

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.

2.2 Substitution in Call-By-Value

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

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

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 2 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 inhabited, it remains inhabited when the variable is substituted by *any* term. This is not correct if there are observably 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* [Lepigre 2016; Wright 1995] when a substitution is involved, *e.g.,*

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

where v is required to be a syntactic value.

2.3 Dependent Elimination in Call-By-Name

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

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

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 [Jaber et al. 2016; Pédrot and Tabareau 2017], dependent elimination is now lost in general. Clearly, if there are observably effectful terms, knowing the behaviour of a predicate on boolean *values* 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 [Pédrot and Tabareau 2017] (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 a 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.

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

This notion of storage operator has been introduced by Krivine in classical realisability [Krivine 1994] 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.

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)

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

2.4 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 [Lepigre 2016].
(3) Effects + substitution + restricted dependent elimination: this is what BTT [Pédrot and Tabareau 2017] is all about.
(4) Effects + substitution + dependent elimination, but inconsistent: the exceptional type theory [Pédrot and Tabareau 2018] 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.

2.5 Explicit Handling of Effects: Call-By-Push-Value

Call-by-push-value [Levy 2001] (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 ranging over values, and data types such as booleans are value types. Call-by-name and call-by-value strategies can then be decomposed into CBPV.

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

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

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

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

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 $[A]^{\mathsf{v}}$, whereas a term of type A is translated as a term of the computation type $\mathcal{F}\ [A]^{\mathsf{v}}$. 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 a value, called *thunkability*.

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

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

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

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

Here, substitution is always valid as a variable of type A is translated as a variable of the value type $\mathcal{U}\ [A]^{\mathsf{n}}$, whereas a term of type A is translated as a term of the computation type $[A]^{\mathsf{n}}$. 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 [Munch-Maccagnoni 2014] and rephrased recently in the context of CBPV by Levy [Levy 2017] (see Section 3.5 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.

2.6 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 [Vákár 2015] and [Ahman et al. 2016], 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 have 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 [Ahman 2018], 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 does this departs from the standard definition of CBPV, but we also claim that this turns out to be a *faux pas* preventing to merely describe 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. This tells us in particular that $\mathcal{F}\ \Box_i^v$ should not be considered convertible to $\Box_i^c$ (and similarly for $\mathcal{U}\ \Box_I^c$ and $\Box_i^v$) as it is true only for some slightly degenerated models.

2.7 A Syntactic Guideline: Weaning and Forcing

We follow the general approach of syntactic models advocated for in [Boulier et al. 2017]. 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 [Jaber et al. 2012] and call-by-name [Jaber et al. 2016] variants, (ii) the weaning translation [Pédrot and Tabareau 2017] 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.

3 DEPENDENT CALL-BY-PUSH-VALUE

In this section, we present an extension of Levy's CBPV [Levy 2001] to dependent types. We coined the name ∂CBPV to avoid confusion with Vákár's dCBPV.

3.1 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 in 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 for readability, as the the separation between values and computations is enforced by typing.

3.2 Meaning of Types

As we have said, there are two classes of types, value types and computation types. Those two classes are reflected respectively by two parallel universe hierarchies $\Box_i^v$ and $\Box_i^c$. Note that the

[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.

values $A, B, v, w ::= \square_i^v \mid \mathcal{U}\ X \mid \Sigma x : A.\ B \mid A + B \mid \mathsf{eq}\ A\ v\ w \mid x \mid \mathsf{thunk}\ t \mid (v, w) \mid \mathtt{inl}\ v \mid \mathtt{inr}\ w \mid \mathtt{refl}$

computations $X, Y, t, u ::= \square_i^c \mid \mathcal{F}\ A \mid \Pi x : A.\ X \mid \mathtt{force}\ t \mid \lambda x : A.\ t \mid t\ v \mid \mathtt{let}\ x : A := t\ \mathtt{in}\ u \mid \mathtt{return}\ v \mid \mathtt{rec}_\Sigma(v, X, t) \mid \mathtt{rec}_+(v, X, t_1, t_2) \mid \mathtt{rec}_{\mathsf{eq}}(v, X, t)$

environments $\Gamma ::= \cdot \mid \Gamma, x : A$

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

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

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

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

$$\frac{\Gamma \vdash_v A : \square_i^v \qquad \Gamma \vdash_v v : A \qquad \Gamma \vdash_v w : A}{\Gamma \vdash_v \mathsf{eq}\ A\ v\ w : \square_i^v} \qquad \frac{\Gamma \vdash_v A : \square_i^v \qquad \Gamma \vdash_v B : \square_j^v}{\Gamma \vdash_v A + B : \square_{\max(i,j)}^v}$$

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

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

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

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

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

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

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

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

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

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

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 $\Gamma \vdash_c t : X$ 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 $\mathtt{eq}\ A\ v_1\ v_2$.

3.3 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 be 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, which is reminiscent of a technique found in [Bizjak et al. 2016]. 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, $\mathtt{let}\ x : A := t\ \mathtt{in}\ X$ would have type $\mathtt{let}\ x : A := t\ \mathtt{in}\ \square_i^c$ which is not a sort of the system. Indeed, as we will see using syntactical models such as forcing or weaning, the rule

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

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 [Ahman 2018].

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.

3.4 Reduction of ∂CBPV

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

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

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

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.

3.5 Unifying Thunkability and Linearity

As we have already mentioned in Section 2, 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 [Levy 2001] after the work of Führmann [Führmann 1999]. 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 [Munch-Maccagnoni 2014] and rephrased recently in the context of CBPV by Levy [Levy 2017]. It semantically expresses the fact that a function is effect-preserving, *e.g.,* by evaluating its arguments first and once. In the works of [Levy 2017; Munch-Maccagnoni 2014], 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 formulation of definitions 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:

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

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 predicates on which substitution can be defined (in call-by-value) and predicates on which dependent elimination can be performed (in call-by-name).

4 ∂CBPV IN ACTION

4.1 Generic Catch For Exceptions

The Exceptional Type Theory [Pédrot and Tabareau 2018] (ExTT) extends type theory with an exception mechanism. In order to remain compatible with standard type theory, ExTT sticks to the call-by-name semantics. This has no consequence on the exception-raising part of the type system, which is materialized by a function $\mathtt{raise} : \Pi A : \square.\, \mathbf{E} \to A$ where $\mathbf{E}$ is the type of exceptions.

Contrastingly, this limits the ability to catch exceptions at the only place where values appear in call-by-name, namely pattern-matching. As such, every inductive type provides a generalized `catch` recursor with an additional clause handling the exception case. For instance, the $\mathtt{catch}_+$ error-handling recursor for the sum type has type

$$\begin{array}{l}\Pi(A\,B : \square)\,(P : (A + B) \to \square).\\ \quad (\Pi x : A.\, P\,(\mathtt{inl}\,x)) \to (\Pi y : B.\, P\,(\mathtt{inr}\,y)) \to (\Pi e : \mathbf{E}.\, P\,(\mathtt{raise}\,(A + B)\,e)) \to\\ \quad\quad \Pi s : A + B.\, P\,s\end{array}$$

and is equipped with the three expected equations, i.e. two for constuctors and one corresponding to catching the exception.

This is fairly obvious in the model justifying ExTT. It is indeed a syntactic translation into CIC which essentially interprets types as pointed types, leaves functions untouched but adds one exceptional constructor to every inductive. As such, the `catch` operator is actually translated as the recursor in the target theory.

Yet, this restriction on exception-catching in ExTT is in stark opposition to what happens in call-by-value, where one can catch exceptions raised from arbitrary thunks. Thankfully, ∂CBPV provides a unified language that allows solving this problem elegantly. That is, in the weaning translation from Section 9 specialized to the case of the error monad, in addition to an exception type $\mathbf{E} : \square_0^v$ and a primitive $\mathtt{raise} : \Pi A : \square_i^v.\, \mathbf{E} \to \mathcal{F}\,A$, it is also possible to precisely express the call-by-value generic catch without losing the dependently-typed flavour of the call-by-name combinator. Namely, there exists a universal `catch` combinator of type

$$\begin{array}{l}\Pi(A : \square_i^v)\,(P : \mathcal{U}\,(\mathcal{F}\,A) \to \square_i^c)).\\ \quad \mathcal{U}\,(\Pi a : A.\, \mathtt{force}\,P\,(\mathtt{thunk}\,(\mathtt{return}\,a))) \to \mathcal{U}\,(\Pi e : \mathbf{E}.\, \mathtt{force}\,P\,(\mathtt{thunk}\,(\mathtt{raise}\,A\,e))) \to\\ \quad\quad \Pi x : \mathcal{U}\,(\mathcal{F}\,A).\, \mathtt{force}\,P\,x\end{array}$$

which internalizes the fact that a computation in $\mathcal{F}\,A$ is either a value or an exception, in the type system itself. Furthermore, it obeys the two expected equations for both values of its argument. From this, it is possible to recover the specialized combinators after performing the call-by-name decomposition from Section 5.

4.2 Proof-Relevant Open Modality

The open modality [Univalent Foundations Program 2013] is used to create new subuniverses inside type-theory in which a mere proposition $\mathbf{P}$ is assumed to hold. Computationally, it corresponds to the so-called *reader* monad, where the type of the cell being read is proof-irrelevant, so as to preserve the good properties of the resulting subuniverse.

Thanks to ∂CBPV, the full reader effect can be brought to dependent type theory, without having to restrict $\mathbf{P}$ to be a mere proposition. Obviously, the price to pay is that there are now observable effects in the resulting theory. The reader effect is given in ∂CBPV by:

- a value type of global cells $\mathbf{P} : \square_0^v$
- a reading primitive $\mathtt{read} : \mathcal{F}\,\mathbf{P}$
- a well-scoped type writing primitive $\mathtt{into} : \mathbf{P} \to \mathcal{U}\,\square_i^c \to \square_i^c$
- a well-scoped writing primitive $\mathtt{enter} : \Pi(p : \mathbf{P})\,(A : \mathcal{U}\,\square_i^c).\, \mathcal{U}\,(\mathtt{force}\,A) \to \mathtt{into}\,p\,A$

$$\texttt{enter}\ p\ (\texttt{thunk}\ \square_i^c)\ A \equiv \texttt{into}\ p\ A \quad \texttt{into}\ p\ (\texttt{thunk}\ \square_i^c) \equiv \square_i^c \quad \texttt{into}\ p\ (\texttt{thunk}\ (\mathcal{F}\ A)) \equiv \mathcal{F}\ A$$
$$\texttt{into}\ p\ (\texttt{thunk}\ (\Pi a : A.\, X)) \equiv \Pi a : A.\, \texttt{into}\ p\ (\texttt{thunk}\ X)$$
$$\texttt{enter}\ p\ (\texttt{thunk}\ (\mathcal{F}\ \mathbf{P}))\ (\texttt{thunk}\ \texttt{read}) \equiv \texttt{return}\ p$$

Fig. 4. Reader equations

subject to equations, the most relevant ones being described in Figure 4.

Intuitively, `into` is a modality that allows to evaluate a type inside a particular value of the global cell, which corresponds logically to relativization of the type argument. The modality definitionally commutes with type formers as described in Figure 4. The two primitives `read` and `enter` respectively access the global cell, and set it locally inside a computation. Contrarily to the usual reader monad, its decomposition as an adjunction forces one to wrap the return type of the `enter` primitive inside that modality. Note nevertheless that, thanks to the modality equations, $\texttt{enter}\ p\ (\texttt{thunk}\ (\mathcal{F}\ A))$ does have type $\mathcal{U}\ (\mathcal{F}\ A) \to \mathcal{F}\ A$ as expected. Through our generalization, it is clear that `enter` corresponds to the return operation of the modality.

As witnessed by the equations, it so happens that `into` is just the specialization of `enter` over the type of types, which is needed to type universes. The remaining equation is the expected interaction between reading and writing.

4.3 Normalization by Evaluation (NbE)

One of the standard ways to implement NbE consists in constructing a presheaf model over a syntactic category of contexts [Abel 2012]. The model is summarily given by the following data:

- A semantic entailment $\Gamma \Vdash t : A$ between a context Γ, a term t and a type A.
- A *soundness* lemma that allows to derive $\Gamma \Vdash t : A$ from $\Gamma \vdash t : A$.
- A syntactic characterization of neutral terms $\Gamma \vdash_{\mathrm{ne}} t : A$ and normal terms $\Gamma \vdash_{\mathrm{nf}} t : A$. While the latter capture the usual notion of terms that cannot be reduced further, the former are a subclass of normal terms that do not trigger new reductions through substitution.
- A two-part *completeness* pair, made of a reflection function $\downarrow_A^\Gamma : \Gamma \vdash_{\mathrm{ne}} n : A \mapsto \Gamma \Vdash n : A$ and a reification function $\uparrow_A^\Gamma : \Gamma \Vdash t : A \mapsto \Gamma \vdash_{\mathrm{nf}} v : A$.

By combining soundness with completeness, one readily obtains an NbE algorithm.

Reasoning in the presheaf topos instead of going through the low-level assembly-like unfolding of presheaf scaffolding, leads to an extremely compact description of the NbE model [Coquand 2019]. Eschewing well-known issues with positive datatypes, it is somewhat folklore that this presheaf construction can be understood as a way to add a well-scoped global environment corresponding to the current context of the proof being built [Jaber et al. 2016]. In other words, presheaves provide a form of side-effects, and the presheaf topos is known as the *direct style* presentation of the resulting programming language. We sketch here how ∂CBPV would provide the syntactic equivalent of the presheaf topos to write an NbE algorithm. The formal definitions of the additional effects at hand are defined in Section 10, but can be conveniently described by the following data, where we omit the expected equations.

- A value type $\texttt{ctx} : \square_0^v$ and a value-returning binary predicate $\subseteq\ (-,-)$ over `ctx`. The `ctx` type stands for contexts of the object language, and is equipped with the expected primitives like context extension and the like, while $\subseteq$ captures context ordering.
- A function $\texttt{here} : \mathcal{F}\ \texttt{ctx}$ returning the current surrounding context.
- A function $\texttt{enter} : \Pi \Gamma : \texttt{ctx}.\ \texttt{let}\ \Delta : \texttt{ctx} := \texttt{here}\ \texttt{in}\ \Gamma \subseteq \Delta \to \mathcal{U}\ (\mathcal{F}\ A) \to \mathcal{F}\ A$ allowing to locally set the current context inside a computation $\mathcal{U}\ (\mathcal{F}\ A)$ provided the new context extends the current one.

This is very similar to the reader monad primitives described above, except for the monotonicity requirement for `enter`. It is now possible to define the NbE components without having to ever refer to the context indexing. For instance, the effectful predicate $\Vdash t : A$ is inductively defined to implicitly refer to the current context and makes use of `enter` to define the arrow case, as:

$$\Vdash t : A \Rightarrow B := \mathcal{U}\ (\Pi x : \mathtt{Tm}\ A. \Vdash x : A \to \mathcal{F}\ (\Vdash \mathsf{app}\ (t, x) : B)).$$

Note how the usual quantification over context extension is encoded by the adjunction. Similarly, reification at an arrow type can be implemented without refering to the ambient context, by simply using `enter`. We will refrain from describing it further as it would lead us into too much detail.

5 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.

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

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

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

This translation is very similar to the call-by-name embedding of simply-typed λ-calculus into CBPV. In particular, $[A \to B]^{\mathsf{n}} := \mathcal{U}\ [A]^{\mathsf{n}} \to [B]^{\mathsf{n}}$ 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 preserves conversion and typing.

Proposition 1 (Substitution). *We have*

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

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

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

5.2 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 then lifting back the resulting value type to a computation type using $\mathcal{F}$.

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

The translation of constructors is analogous:

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

Fig. 5. CBN translation of recursors for coproducts

$$[\mathtt{inl}\, t]^{\mathsf{n}} := \mathtt{return}\ (\mathtt{inl}\ (\mathtt{thunk}\ [t]^{\mathsf{n}})) \quad [\mathtt{inr}\, t]^{\mathsf{n}} := \mathtt{return}\ (\mathtt{inr}\ (\mathtt{thunk}\ [t]^{\mathsf{n}})).$$

The translation of the recursor is more problematic as it requires one to recover the value out of the computation by using a `let` binder. As mentionned in Section 2.3, 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:

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

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

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

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

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

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 Figure 5. To prove an extension of Proposition 2 to BTT, the main point is to check that $[\mathtt{drec}_+(t, X, u_1, u_2)]^{\mathsf{n}}$ has a type convertible to $[\sigma_{A+B}\ v\ X]^{\mathsf{n}}$. 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`.

5.3 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

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

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

However, linearity of a predicate is a semantic notion and is undecidable in general. Ahman [Ahman 2018] 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.

6 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.

6.1 Call-By-Value Translation in CBPV

The standard by-value translation interprets types A as value types $[\![A]\!]^{\mathsf{v}}$ (in particular $[\![\square_i]\!]^{\mathsf{v}} \equiv \square_i^v$), and terms $t : A$ as computations of $[t]^{\mathsf{v}} : \mathcal{F}\ [\![A]\!]^{\mathsf{v}}$. But from $A : \square_i$, we only get $[A]^{\mathsf{v}} : \mathcal{F}\ \square_i^v$, and there is no way to define $[\![A]\!]^{\mathsf{v}} : \square_i^v$ as using a `let` binding can only produce a computation type and not a value type (we come back to this problem in Section 6.2). To solve this, we need to define $[\![A]\!]^{\mathsf{v}}$ primitively to $[A]^{\mathsf{v}}$, 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* $\Gamma \vdash_{\mathsf{CIC}^v} A : \square$ *then* A *is a syntactic value.*

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).

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

We do not detail the typing rules of CIC^v as we do not want to dwell too much on it. Figure 6 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 $[\![v]\!]^{\mathsf{v}}$ and a term t as $[t]^{\mathsf{v}}$.

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

Fig. 6. Call-by-value type theory

Definition 9 (By-value translation)**.** The by-value translation is defined in Figure 7.

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

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

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

6.2 Limitation of Call-By-Value

The theory induced 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.

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

Fig. 7. Call-by-value translation

6.3 Recovering CIC through Thunkability

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

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

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 $[A]^{\mathsf{v}}$ 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.

7 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 6.
- Second, we restrict the computations to be observationally pure, by requiring them to preserve thunkability everywhere.

This technique is similar to Girard's boring translation [Girard 1987] in linear logic. As it will turn out in Section 10, 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 8.

This extension can be intuitively described as follows.

- The type $\mathcal{E}\ell\, X$ is equal to $\mathtt{let}\ A : \Box_i^v := X\ \mathtt{in}\ \mathcal{F}\ A$ but is defined only for thunkable types. Its equation on values allows transparently considering elements of $\mathcal{F}\ A$ as living in $\mathcal{E}\ell\ (\mathtt{return}\, A)$.
- The judgment $\Gamma \vdash t$ th. encodes the fact that a computation $\Gamma \vdash t : \mathcal{E}\ell\, X$ satisfies the thunkability equations when it makes sense typing-wise. Returned values are in particular thunkable, and thunkability is preserved by constructions not introducing effects.

$$
\begin{array}{lcl}
A, B, v, w & ::= & \ldots \mid \mathtt{El}\, X \mid \Theta x : A.\, B \mid \theta x : A.\, v \mid t \cdot w \mid \Uparrow t \\
X, Y, t, u & ::= & \ldots \mid \mathcal{El}\, X \mid \Downarrow v
\end{array}
$$

$$
\frac{\Gamma \vdash X : \mathcal{F}\, \square_i^v \qquad \Gamma \vdash X \text{ th.}}{\Gamma \vdash \mathtt{El}\, X : \square_i^v} \qquad \frac{\Gamma \vdash X : \mathcal{F}\, \square_i^v \qquad \Gamma \vdash X \text{ th.}}{\Gamma \vdash \mathcal{El}\, X : \square_i^c}
$$

$$
\frac{\Gamma \vdash t : \mathcal{El}\, X \qquad \Gamma \vdash t \text{ th.}}{\Gamma \vdash \Uparrow t : \mathtt{El}\, X} \qquad \frac{\Gamma \vdash v : \mathtt{El}\, X}{\Gamma \vdash \Downarrow v : \mathcal{El}\, X} \qquad \frac{\Gamma \vdash v : \mathtt{El}\, X}{\Gamma \vdash \Downarrow v \text{ th.}}
$$

$$
\frac{\Gamma \vdash A : \square_i^v \qquad \Gamma, x : A \vdash B : \mathtt{El}\, (\mathtt{return}\, \square_j^v)}{\Gamma \vdash \Theta x : A.\, B : \square^v_{\max(i,j)}} \qquad \frac{\Gamma, x : A \vdash v : \mathtt{El}\, (\Downarrow B)}{\Gamma \vdash \theta x : A.\, v : \Theta x : A.\, B}
$$

$$
\frac{\Gamma \vdash t : \Theta x : A.\, B \qquad \Gamma \vdash w : A}{\Gamma \vdash t \cdot w : \mathtt{El}\, (\Downarrow B\{x := w\})}
$$

$$
\frac{\Gamma \vdash v : A}{\Gamma \vdash \mathtt{return}\, v \text{ th.}} \qquad \frac{\Gamma \vdash t \text{ th.} \qquad \Gamma, x : A \vdash u \text{ th.}}{\Gamma \vdash \mathtt{let}\, x : A := t\, \mathtt{in}\, u \text{ th.}} \qquad \frac{\Gamma \vdash t \text{ th.} \qquad \Gamma, x : A \vdash u \text{ th.}}{\Gamma \vdash \mathtt{dlet}\, x : A := t\, \mathtt{in}\, u \text{ th.}}
$$

$$
\mathtt{let}\, x : A := \Downarrow t\, \mathtt{in}\, \Phi\{\mathtt{return}\, x\} \equiv \Phi\{\Downarrow t\} \qquad \Downarrow \Uparrow t \equiv t \qquad (\theta x : A.\, v) \cdot w \equiv v\{x := w\}
$$

$$
\mathcal{El}\, (\mathtt{return}\, A) \equiv \mathcal{F}\, A
$$

Fig. 8. Thunkable types

- The type $\mathtt{El}\, X$ captures the subset type $\{v : \mathcal{U}\, (\mathcal{El}\, X) \mid \cdot \vdash \mathtt{force}\, v \text{ th.}\}$ provided X is thunkable itself so that the expression typechecks. One can project out the underlying element of $\mathcal{El}\, X$ with $\Downarrow -$ (which corresponds to `force` $-$), and embed values with $\Uparrow -$ (which corresponds to `thunk` $-$).
- The type $\Theta x : A.\, B$ morally stands for $\Pi x : A.\, \mathtt{El}\, (\Downarrow B)$, but this type does not make sense in ∂CBPV as the codomain is a value type. This type is more correctly described as functions $f : \Pi x : A.\, \mathcal{F}\, B_0$ that preserve thunkability pointwise when $B \equiv \Uparrow (\mathtt{return}\, B_0)$.

Note that the thunkability property does not appear as a part of the proof term and needs to be reconstructed out of thin air. As such, when describing the thunkability extension in our syntactic translations, we will always assume that our target theory will be some extensional flavour of CIC, i.e. where propositional equality can be reflected as definitional equality. It seems unlikely to be able internalize thunkability in an intensional type theory without diving into a coherence hell.

7.1 Negative Fragment

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

$$
\begin{array}{lcl}
[\![A]\!]^\theta & := & \mathtt{El}\, (\Downarrow [A]^\theta) \\
[\square_i]^\theta & := & \Uparrow \mathtt{return}\, \square_i^v \\
[\Pi x : A.\, B]^\theta & := & \Uparrow \mathtt{return}\, (\Theta x : [\![A]\!]^\theta.\, [B]^\theta) \\
[x]^\theta & := & x \\
[t\ u]^\theta & := & \Uparrow \mathtt{let}\, f := \Downarrow [t]^\theta\, \mathtt{in}\, \Downarrow (f \cdot [u]^\theta) \\
[\lambda x : A.\, t]^\theta & := & \Uparrow \mathtt{return}\, (\theta x : [\![A]\!]^\theta.\, [t]^\theta)
\end{array}
$$

As explained above, assuming $\mathtt{El}\, (\mathtt{return}\, A)$ is a semantic subtype of $\mathcal{U}\, (\mathcal{F}\, A)$, 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 $[t\{x := r\}]^\theta \equiv_{\partial\mathsf{CBPV}} [t]^\theta\{x := [r]^\theta\}$.*

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 $[\![\Gamma]\!]^\theta \vdash [t]^\theta : [\![A]\!]^\theta$.*

Proof. By induction on the typing derivation. This proof relies crucially on thunkability. □

7.2 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.

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

where

- $\tilde{P} := \lambda s.\, \mathcal{E}\ell\, (\Downarrow [P\ s_0]^\theta)\{s_0 := \Uparrow (\mathtt{return}\ s)\}$
- $\tilde{u}_i := \lambda x. \Downarrow [u_i\ x_0]^\theta\{x_0 := \Uparrow (\mathtt{return}\ x)\}$

Proposition 10. *Assuming ∂CBPV enjoys the η-rule $\Uparrow (\Downarrow t) \equiv t$ and that recursors preserve thunkability, then $[-]^\theta$ 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.

$$
\begin{array}{lcl}
\tilde{u}_1 & : & \Pi x : [\![A]\!]^\theta.\, \tilde{P}\ (\mathtt{inl}\ x) \\
\tilde{u}_2 & : & \Pi y : [\![B]\!]^\theta.\, \tilde{P}\ (\mathtt{inr}\ y)
\end{array}
$$

- second that its type is convertible to the one of full dependent elimination, i.e.

$$\mathtt{let}\ s := \Downarrow [s]^\theta\ \mathtt{in}\ \tilde{P}\ s \equiv \mathcal{E}\ell\, (\Downarrow [P\ s]^\theta)$$

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.

8 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.

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

Fig. 9. Reader Translation

8.1 Translation

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 9.

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:

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

where we write

$$\tilde{X} := \lambda s.\ [X]\ p\ s \qquad \text{and} \qquad \tilde{u_i} := \lambda x.\ [u_i]\ p\ x.$$

Soudness is easily showed to be preserved by this extension.

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

8.2 Reader Effects

We implement the primitive described in Section 4.2 in Figure 10. It is easy to check that the equations given previously for those primitives hold.

8.3 A Glimpse of BTT

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.

$$
\begin{array}{lcl}
[\mathbf{P}] & : & [\![\square_0^v]\!]^v \\
 & := & \mathbb{P} \\
[\mathtt{read}] & : & [\![\mathcal{F}\ \mathbf{P}]\!]^c \\
 & := & \lambda p : \mathbb{P}.\, p \\
[\mathtt{into}] & : & [\![\mathbf{P} \to \mathcal{U}\ \square_i^c \to \square_i^c]\!]^c \\
 & := & \lambda(\_ : \mathbb{P})\ (p : \mathbb{P})\ (A : \mathbb{P} \to \square_i).\, A\ p \\
[\mathtt{enter}] & : & [\![\Pi(p : \mathbf{P})\ (A : \mathcal{U}\ \square_i^c).\, \mathcal{U}\ (\mathtt{force}\ A) \to \mathtt{into}\ p\ A]\!]^c \\
 & := & \lambda(\_ : \mathbb{P})\ (p : \mathbb{P})\ (A : \mathbb{P} \to \square_i)\ (x : \Pi p : \mathbb{P}.\, A\ p).\, x\ p
\end{array}
$$

Fig. 10. Reader Effects

Example 2. Unfolding the translations, implementing dependent elimination for coproducts in $[[-]^n]$ 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.

8.4 Thunkability

The reader translation also models the extensions needed for the call-by-thunkable interpretation. In the remainder of this section, we assume that the target theory enjoys extensionality.

Proposition 11. *A term $t : [\![\mathcal{F}\ A]\!]^c \equiv \mathbb{P} \to [\![A]\!]^v$ satisfies the thunkable equation through the reader translation iff $p, q : \mathbb{P} \vdash t\ p \equiv t\ q$.*

In particular, if $X : \mathcal{F}\ \square_i^v$ is thunkable, it is possible to extend this semantic thunkability to inhabitants of $\mathcal{E}\ell\ X := \mathtt{let}\ A : \square_i^v := X\ \mathtt{in}\ \mathcal{F}\ A$ by defining thunkability of any $t : \mathcal{E}\ell\ X$ as

$$\theta_X\ t := \Pi(p\ q : \mathbb{P}).\, [t]\ p = [t]\ q.$$

We insist on the crucial fact that this equation is only well-typed if X is thunkable itself, as the left-hand side has type $[X]\ p$ when the right-hand side has type $[X]\ q$.

All the other constructions are defined as described intuitively in Section 7, e.g.

$$[\mathtt{El}\ X] := \Sigma x : [\![\mathcal{E}\ell\ X]\!]^c.\, \theta_X\ x$$

and similarly $\Theta x : A.\, B$ is the subtype of functions that are pointwise thunkable.

Proposition 12. *The reader translation is a model of call-by-thunkable.*

Interestingly, the call-by-thunkable translation is *not* trivially isomorphic to a pure theory. Terms in $\mathtt{El}\ X$ can use the $\mathbb{P}$ argument and remain thunkable, as long as they use it in a proof-irrelevant way. This is surprisingly similar to the restriction of the open modality to mere propositions.

9 WEANING TRANSLATION IN ∂CBPV

9.1 Weaning Translation

This section describes the weaning translation from ∂CBPV into CIC, based on the notion of *self-algebraic proto-monad*. The weaning translation corresponds to a call-by-name variant of Moggi's monadic translation [Moggi 1991] in a dependent setting. In the same way as in Moggi's work, the coarse idea is to interpret computations in terms of algebras for a monad T in a pure type theory. However, Moggi's interpretation considered free algebras (that is type of the form $\mathsf{T}\ A$) which amounts in categorical language to considering the Kleisli category induced by the monad. In a dependent setting, it appears to the other extreme case, considering every algebra, is more suitable. This corresponds to the Eilenberg-Moore category induced by the monad. In a dependent setting, the monadic interpretation requires more structure on the monad under consideration. Because the universe of computation types is reflected in the theory, the universe of algebra must itself be an algebra for the monad, a situation that we coined *self-algebraic monad*. Note that because we don't need all the monadic laws usually available for a monad, we talk here about *proto-monad* instead.

In this section, we directly rephrase the weaning translation in the setting of ∂CBPV whereas in the original paper, the translation is based on BTT. The weaning translation can be instantiated for instance to describe dynamic exceptions, non-determinism or non-termination. See [Pédrot and Tabareau 2017] for a more in-depth description of the translation and its various applications.

Definition 15 ([Pédrot and Tabareau 2017])**.** A self-algebraic proto-monad is given by the following universe-polymorphic family of CIC terms:

- $\mathsf{T} : \square_i \to \square_i$
- $\mathtt{ret} : \Pi A : \square_i.\ A \to \mathsf{T}\ A$
- $\mathtt{bnd} : \Pi (A : \square_i)\ (B : \square_j).\ \mathsf{T}\ A \to (A \to \mathsf{T}\ B) \to \mathsf{T}\ B$
- $\mathtt{El} : \mathsf{T}\ \boxbox_i \to \boxbox_i$
- $\mathtt{hbnd} : \Pi (A : \square_i)\ (B : \mathsf{T}\ \boxbox_j).\ \mathsf{T}\ A \to (A \to (\mathtt{El}\ B).\pi_1) \to (\mathtt{El}\ B).\pi_1$

where

$$\boxbox_i := \Sigma A : \square_i.\ \mathsf{T}\ A \to A$$

furthermore subject to the following definitional equations:

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

For brevity, we will use implicit type arguments for the monadic combinators. Similarly, when forming inhabitants of $\boxbox$ in the translation below, we will omit the algebra morphism and simply write $\{\!\{A\}\!\} : \boxbox$ given any $A : \square$. The morphisms are the same as in our previous paper [Pédrot and Tabareau 2017] and are canonical, e.g. for a type of the form $\mathsf{T}\ A$, 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 11. 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 13 (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 [Pédrot and Tabareau 2017]. □

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

Fig. 11. Weaning Translation

$$
\begin{array}{lcl}
[\mathbf{E}] & : & [\![\square_0^v]\!]^v \\
 & := & \mathbb{E} \\
[\mathtt{raise}] & : & [\![\mathbf{E} \to \Pi A : \square_i^v.\, \mathcal{F}\ A]\!]^c \\
 & := & \lambda(e : \mathbb{E})\ (A : \square_i).\, \mathtt{inr}\ e \\
[\mathtt{catch}] & : & [\![\Pi(A : \square_i^v)\ (P : \mathcal{U}\ (\mathcal{U}\ (\mathcal{F}\ A) \to \square_i^c)). \\
 & & \quad \mathcal{U}\ (\Pi a : A.\, \mathtt{force}\ P\ (\mathtt{thunk}\ (\mathtt{return}\ a))) \to \\
 & & \quad \mathcal{U}\ (\Pi e : \mathbf{E}.\, \mathtt{force}\ P\ (\mathtt{thunk}\ (\mathtt{raise}\ A\ e))) \to \\
 & & \quad \Pi x : \mathcal{U}\ (\mathcal{F}\ A).\, \mathtt{force}\ P\ x]\!]^c \\
 & := & \lambda(A : \square_i)\ (P : (A + \mathbb{E}) \to \mathsf{T}\ \square_i). \\
 & & \quad \lambda(u_v : \Pi a : A.\, (\mathtt{El}\ (P\ (\mathtt{inl}\ a))).\pi_1)\ (u_e : \Pi e : \mathbb{E}.\, (\mathtt{El}\ (P\ (\mathtt{inr}\ e))).\pi_1)\ (\hat{x} : A + \mathbb{E}). \\
 & & \quad\quad \mathtt{rec}_+(\hat{x}, \lambda x.\, (\mathtt{El}\ (P\ x)).\pi_1, u_v, u_e)
\end{array}
$$

Fig. 12. Exceptional effects

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

9.2 Effectful Primitives

Through this model, it is possible to extend ∂CBPV with all the new primitives coming from the T type constructor. For instance, any free algebraic monad is also a self-algebraic one trivially. In this case, any constructor from this monad is straightforwardly reflected as a new parameterized computation of type $\mathcal{F}\ A$ for some A, and the induction principle over T is reflected as a kind of generalized try-catch block. For the sake of simplicity, we consider $\mathsf{T}\ A := A + \mathbb{E}$ for some exception type $\mathbb{E}$ in the target theory, with $\mathtt{El}\ (\mathtt{inl}\ X) := X$ and $\mathtt{El}\ (\mathtt{inr}\ e) := \Omega\ e$ for some arbitrary $\Omega : \mathbb{E} \to \square$, and we implement the primitives from Section 4 in Figure 12.

The translation is slightly cluttered due to the fact that computation types are algebras, but apart from that the implementation is fairly simple. Raising an exception corresponds to using the right injection of the sum type, and the generalized try-catch block uses dependent elimination over the monadic value to dispatch the corresponding branch.

9.3 Thunkability

Thunkability is not very interesting in this model. Assuming extensionality, we have indeed:

Proposition 14. *A term* $t : [[\mathcal{F}\ A]]^c \equiv \mathsf{T}\ [[A]]^v$ *is thunkable iff there exists* $v : [[A]]^v$ *s.t.* $t \equiv \mathtt{ret}\ v$.

The weaning translation does provide the required extensions for the call-by-thunkable decomposition, but the resulting model is essentially the target theory, as thunkability prevent any effect to creep in.

10 FORCING TRANSLATION IN ∂CBPV

This section is devoted to the definition of the forcing model of ∂CBPV. This model is essentially the one from [Jaber et al. 2012] and bears similarity with a CBPV model described by Levy [Levy 2001, 2002] which is a state-like, simply-typed variant of it. Contrarily to the other ∂CBPV models presented in this paper, we require the target theory, henceforth refered to as ECIC, to be extensional, i.e. propositional and definitional equality must coincide. This requirement is slightly stronger than [Jaber et al. 2012] to be able to interpret ∂CBPV conversion by conversion in the target.

The forcing translation described in [Jaber et al. 2012] can be seen as a formalisation of the presheaf construction in category theory. Given a base category $\mathbb{P}$, whose objects are traditional referred to as forcing conditions, and an initial forcing condition p, a computation type is interpreted as a family of types indexed by the forcing conditions q that can be reached from p, with a monotonicity requirement. In Kripke terminology, one says that q is in the future of p. In a categorical setting, this corresponds to functors from $\mathbb{P}$ to the category of sets. The rest of the translation is performed in the same way, reflecting the categorical machinery of presheaves. In particular, the translation of dependent product comes with an external equality requirement which reflects the naturality condition of natural transformations.

In ∂CBPV, we can split the translation in two steps. First a translation which does not mentioned the naturality condition and then the definition of what it means to be thunkable in this setting—which corresponds precisely to the naturality condition in the presheaf construction. Therefore, the translation of CIC described in [Jaber et al. 2012] can be recovered by combining the forcing translation on ∂CBPV with the call-by-thunkable translation of Section 7.

10.1 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.

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

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 $(q : \mathbb{P})\ (\alpha : q \leq p)$.

$$
\begin{array}{lcl}
[\![A]\!]^\Gamma_p & := & [A]^\Gamma_p.\mathtt{type}\ p\ \mathbf{1}_p \\
[\Box^c_i]^\Gamma_p & := & \Box_i \\
[\Box^v_i]^\Gamma_p & := & \{\!\{\lambda(q\,\alpha : p).\, \Box_i\ q\}\!\} \\
[\Pi x : A.\, X]^\Gamma_p & := & \Pi x : [\![A]\!]^\Gamma_p.\, [X]^\Gamma_p \\
[\mathcal{U}\ X]^\Gamma_p & := & \{\!\{\lambda(q\,\alpha : p).\, \Pi(r\,\beta : q).\ (\beta * \alpha) \bullet_\Gamma [X]^\Gamma_r\}\!\} \\
[\mathcal{F}\ A]^\Gamma_p & := & [\![A]\!]^\Gamma_p \\
[\![\cdot]\!]_p & := & p : \mathbb{P} \\
[\![\Gamma, x : A]\!]_p & := & [\![\Gamma]\!]_p, x : [\![A]\!]^\Gamma_p
\end{array}
$$

$$
\begin{array}{lcl}
[x]^\Gamma_p & := & x \\
[\mathtt{thunk}\ t]^\Gamma_p & := & \lambda(q\,\alpha : p).\, \alpha \bullet_\Gamma [t]^\Gamma_q \\
[\mathtt{force}\ v]^\Gamma_p & := & [v]^\Gamma_p\ p\ \mathbf{1}_p \\
[\lambda x : A.\, t]^\Gamma_p & := & \lambda x : [\![A]\!]^\Gamma_p.\, [t]^{\Gamma, x:A}_p \\
[t\ v]^\Gamma_p & := & [t]^\Gamma_p\ [v]^\Gamma_p \\
 & & \\
[\mathtt{let}\ x : A := t\ \mathtt{in}\ u]^\Gamma_p & := & (\lambda x : [\![A]\!]_p.\, [u]^{\Gamma, x:A}_p)\ [t]^\Gamma_p \\
[\mathtt{dlet}\ x : A := t\ \mathtt{in}\ u]^\Gamma_p & := & (\lambda x : [\![A]\!]_p.\, [u]^{\Gamma, x:A}_p)\ [t]^\Gamma_p \\
[\mathtt{return}\ v]^\Gamma_p & := & [v]^\Gamma_p
\end{array}
$$

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

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

Fig. 13. Forcing Translation

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

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

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

Definition 19 (Forcing translation)**.** The forcing translation, defined at Fig. 13, is indexed by a ∂CBPV environment Γ and a condition p and produces an ECIC term.

Let us walk through this definition. Computation types are interpreted as mere types, while value types are interpreted as *monotonic* types. In particular as $\Box^v_i : \Box^v_{i+1}$, 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 15. *If* $[\![\Gamma]\!]_p \vdash_{\mathrm{ECIC}} t : A$*, then* $\mathbf{1}_p \bullet_\Gamma t \equiv t$ *and* $(\beta * \alpha) \bullet_\Gamma t \equiv \alpha \bullet_\Gamma (\beta \bullet_\Gamma t)$ *assuming well-typedness of morphisms.*

Proposition 16 (Typing soundness). *Assume* $\Gamma \vdash_c t : X$, *then* $[\![\Gamma]\!]_p \vdash [t]^\Gamma_p : [X]^\Gamma_p$ *and similarly for values.*

Proposition 17 (Computational soundness). *For all* $\Gamma \vdash_c t,u : A$, *if* $t \equiv u$ *then* $[t]^\Gamma_p \equiv [u]^\Gamma_p$ *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.

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

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 [Jaber et al. 2016].

10.2 Thunkability

What would the thunkable translation look like? To answer this, we need to peek at the forcing translation of thunkability. Without further ado:

Proposition 18. *A term* $\Gamma \vdash_{\partial\mathrm{CBPV}} t : \mathcal{F}\, A$ *is thunkable iff it is natural, i.e. for all* $p, q, \alpha : q \leq p$,

$$\alpha \bullet_\Gamma [t]^\Gamma_q \equiv_{\mathrm{ECIC}} \alpha \circ^\Gamma_A [t]^\Gamma_p.$$

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

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

To the best of our knowledge, this property had never been observed as such in the literature, even though it is far-reaching. The closest thing can be found in Levy's state-like model [Levy 2002], where naturality is part of the *definition* of the model, rather than a property arising *a posteriori* from a generic equation unrelated to the model at hand.

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 quite an 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 an intensional CIC, although the extension of CIC with strict propositions [Gilbert et al. 2019] seems to bring such a model into the realm of possibility.

11 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 papers which were the source for the models of ∂CBPV [Jaber et al. 2016; Pédrot and Tabareau 2017, 2018], there are two lines of work that are particularly close to the current paper, namely Ahman's [Ahman 2018] and Vákár's [Vákár 2015, 2017].

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.

[Pédrot and Tabareau 2017] already advocated against Vákár's dCBPV. For the sake of self-containedness, we summarize their 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 6. 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. [Bowman et al. 2018] 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 [Lepigre 2016]. 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

Andreas Abel. 2012. *Normalization by Evaluation, Dependent Types and Impredicativity.* Ph.D. Dissertation. Institut für Informatik Ludwig-Maximilians-Universität München.

Danel Ahman. 2018. Handling Fibred Algebraic Effects. *Proc. ACM Program. Lang.* 2, POPL, Article 7 (Jan. 2018), 29 pages. https://doi.org/10.1145/3158095

Danel Ahman, Neil Ghani, and Gordon D. Plotkin. 2016. Dependent Types and Fibred Computational Effects. In *19th International Conference on Foundations of Software Science and Computation Structures*. Springer Berlin Heidelberg, Eindhoven, The Netherlands, 36–54. https://doi.org/10.1007/978-3-662-49630-5_3

Gilles Barthe and Tarmo Uustalu. 2002. CPS Translating Inductive and Coinductive Types. In *Proceedings of Partial Evaluation and Semantics-based Program Manipulation*. ACM, 131–142.

Ales Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus Ejlers Møgelberg, and Lars Birkedal. 2016. Guarded Dependent Type Theory with Coinductive Types. In *Foundations of Software Science and Computation Structures - 19th International Conference, FOSSACS 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 (Lecture Notes in Computer Science)*, Bart Jacobs and Christof Löding (Eds.), Vol. 9634. Springer, 20–35. https://doi.org/10.1007/978-3-662-49630-5_2

Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. 2017. The Next 700 Syntactical Models of Type Theory. In *Proceedings of Certified Programs and Proofs*. ACM, 182–194.

William J. Bowman, Youyou Cong, Nick Rioux, and Amal Ahmed. 2018. Type-preserving CPS translation of Σ and Π types is not not possible. *PACMPL* 2, POPL (2018), 22:1–22:33. https://doi.org/10.1145/3158110

Thierry Coquand. 2019. Canonicity and normalization for dependent type theory. *Theor. Comput. Sci.* 777 (2019), 184–191. https://doi.org/10.1016/j.tcs.2019.01.015

Carsten Führmann. 1999. Direct Models of the Computational Lambda-calculus. *Electronic Notes in Theoretical Computer Science* 20 (1999), 245 – 292. https://doi.org/10.1016/S1571-0661(04)80078-1 MFPS XV, Mathematical Foundations of Progamming Semantics, Fifteenth Conference.

Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. 2019. Definitional proof-irrelevance without K. *PACMPL* 3, POPL (2019), 3:1–3:28. https://doi.org/10.1145/3290316

Jean-Yves Girard. 1987. Linear Logic. *Theor. Comput. Sci.* 50 (1987), 1–102. https://doi.org/10.1016/0304-3975(87)90045-4

V. Glivenko. 1929. Sur Quelques Points de la Logique de M. Brouwer. *Bulletins de la classe des sciences* 15 (1929), 183–188.

Timothy Griffin. 1990. 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*. 47–58. https://doi.org/10.1145/96709.96714

Hugo Herbelin. 2005. On the Degeneracy of Sigma-Types in Presence of Computational Classical Logic. In *Seventh International Conference, TLCA '05, Nara, Japan. April 2005, Proceedings (Lecture Notes in Computer Science)*, Pawel Urzyczyn (Ed.), Vol. 3461. Springer, 209–220.

Guilhem Jaber, Gabriel Lewertowski, Pierre-Marie Pédrot, Matthieu Sozeau, and Nicolas Tabareau. 2016. 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*. 367–376.

Guilhem Jaber, Nicolas Tabareau, and Matthieu Sozeau. 2012. Extending Type Theory with Forcing. In *LICS 2012 : Logic In Computer Science*. Dubrovnik, Croatia, 0–0.

Jean-Louis Krivine. 1994. Classical Logic, Storage Operators and Second-Order lambda-Calculus. *Ann. Pure Appl. Logic* 68, 1 (1994), 53–78. https://doi.org/10.1016/0168-0072(94)90047-7

Rodolphe Lepigre. 2016. 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*. 476–502. https://doi.org/10.1007/978-3-662-49498-1_19

Paul Blain Levy. 2001. *Call-by-push-value*. Ph.D. Dissertation. Queen Mary, University of London.

Paul Blain Levy. 2002. Possible World Semantics for General Storage in Call-By-Value. In *Computer Science Logic, 16th International Workshop, CSL 2002, 11th Annual Conference of the EACSL, Edinburgh, Scotland, UK, September 22-25, 2002, Proceedings*. 232–246. https://doi.org/10.1007/3-540-45793-3_16

Paul Blain Levy. 2017. Contextual Isomorphisms. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 400–414. https://doi.org/10.1145/3009837.3009898

Per Martin-Löf. 2006. 100 years of Zermelo's axiom of choice: what was the problem with it? *Comput. J.* 49, 3 (2006), 345–350. https://doi.org/10.1093/comjnl/bxh162

Eugenio Moggi. 1991. Notions of Computation and Monads. *Information and Computation* 93, 1 (July 1991), 55–92.

Guillaume Munch-Maccagnoni. 2014. Models of a Non-Associative Composition. In *17th International Conference on Foundations of Software Science and Computation Structures*, Anca Muscholl (Ed.), Vol. 8412. Springer, Grenoble, France, 396–410.

Pierre-Marie Pédrot and Nicolas Tabareau. 2017. 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, 1–12. https://doi.org/10.1109/LICS.2017.8005113

Pierre-Marie Pédrot and Nicolas Tabareau. 2018. 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. 245–271. https://doi.org/10.1007/978-3-319-89884-1_9

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study.

Matthijs Vákár. 2015. A Framework for Dependent Types and Effects. arXiv:arXiv:1512.08009 https://arxiv.org/abs/1512.08009 draft.

Matthijs Vákár. 2017. *In Search of Effectful Dependent Types*. Ph.D. Dissertation. University of Oxford.

Andrew Wright. 1995. Simple Imperative Polymorphism. In *LISP and Symbolic Computation*. 343–356.