An Effectful Way to Eliminate Addiction to Dependence

Pierre-Marie Pédrot, Nicolas Tabareau

An Effectful Way to Eliminate Addiction to Dependence

Pierre-Marie Pédrot
University of Ljubljana
pierre-marie.pedrot@fmf.uni-lj.si

Nicolas Tabareau
INRIA
nicolas.tabareau@inria.fr

***Abstract*—We define a monadic translation of type theory, called weaning translation, that allows for a large range of effects in dependent type theory—such as exceptions, non-termination, non-determinism or writing operation. Through the light of a call-by-push-value decomposition, we explain why the traditional approach fails with type dependency and justify our approach. Crucially, the construction requires that the universe of algebras of the monad forms itself an algebra. The weaning translation applies to a version of the Calculus of Inductive Constructions (CIC) with a restricted version of dependent elimination. Finally, we show how to recover a translation of full CIC by mixing parametricity techniques with the weaning translation. This provides the first effectful version of CIC.**

I. Introduction

The gap between type theories such as CIC and mainstream programming languages comes to a large extend from the absence of effects in type theories, because of its complex interaction with dependency. For instance, it has already been noticed that inductive types and dependent elimination do not scale well to CPS translations and classical logic [1], [2]. Furthermore, the traditional way to integrate effects in functional programming using monads does not scale to dependency because the monad leaks in the type during substitution.

In this paper, we propose Baclofen Type Theory (BTT), a stripped-down version of CIC, and we provide generic notion of syntactic models[1] of it that allows for a large range of effects in dependent type theory—exceptions, non-termination, non-determinism or writing operation. BTT has a restricted version of dependent elimination to overcome the difficulty to marry effect and dependency. The syntactic models are given by the *weaning translation* of BTT into CIC, using a variant of the traditional monadic translation. The need for this variant can be explained by analyzing the call-by-push-value (CBPV) decomposition of call-by-value and call-by-name reduction strategies. Crucially, our construction requires that the universe of algebras of the monad forms itself an algebra. A monad satisfying this property is said to be *self-algebraic*. We then show how very common monads satisfy this property and thus give rise to effects that can be integrated to BTT.

Finally, by mixing parametricity techniques with the weaning translation, we show how to recover a translation of full CIC, giving rise to the first effectful version of CIC.

[1]By syntactic models, we mean a model directly expressed in a type theory through a program transformation, as advocated in [3].

Plan of the paper.

In Section II, we explain the main points of the construction through the CBPV decomposition. Then, Section III and IV describe the weaning translation for self-algebraic proto-monads on BTT. Section V describes various instances of self-algebraic proto-monads and their asociated effects. Section VI presents a linearity condition to ease the use of BTT on non-recursive inductive types and finally Section VII explains how a mild modification of the weaning translation using parametricity techniques allows one to recover a translation of full CIC.

Plugin implementation.

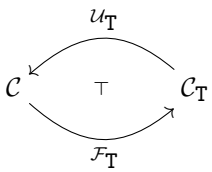
As it is the case for other syntactic models [4], [3], it is possible to implement the weaning translation as a Coq plugin. The plugin is available at `https://github.com/CoqHott/coq-effects`.

II. Genesis of the construction

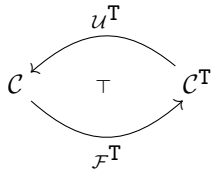
This section presents a global overview of the weaning translation given in this paper. It is based on two key ingredients that allow us to define a monadic translation of CIC for a large range of effects: i) the use of the call-by-name decomposition in Levy's call-by-push-value [5] (CBPV) and ii) its instantiation with Eilenberg-Moore algebras for self-algebraic monads.

A. The two canonical decompositions of a monad.

The use of monads to interpret effects in functional programming languages comes back from the seminal work of Moggi [6]. In its traditional view, the monadic interpretation amounts to consider functions from A to B as functions of type $A \to \mathtt{T}\ B$ where $\mathtt{T}$ is a computational monad. From a categorical point of view, this interpretation consists in working in the Kleisli category $\mathcal{C}_\mathtt{T}$ induced by a monad $\mathtt{T}$ on $\mathcal{C}$, where the objects are those of $\mathcal{C}$ and the morphisms of $\mathcal{C}_\mathtt{T}(A, B)$ are given by $\mathcal{C}(A, \mathtt{T}B)$. Actually, the Kleisli category is one of the two canonical notions of category of computations that is part of a left and right adjoints decomposition of the monad,



where $\mathcal{F}_{\mathtt{T}}$ is the identity on objects, $\mathcal{U}_{\mathtt{T}}$ is equal to $\mathtt{T}$ on objects and $\mathcal{U}_{\mathtt{T}} \circ \mathcal{F}_{\mathtt{T}} = \mathtt{T}$. There is another canonical decomposition where the category of computations is given by algebras of the monad, *i.e.,* objects A with an arrow $\mathtt{T}\ A \to A$ compatible with the monadic operations. The category of algebras $\mathcal{C}^{\mathtt{T}}$, called the Eilenberg-Moore category, is part of the adjunction



where $\mathcal{F}^{\mathtt{T}}$ is equal to $\mathtt{T}$ on objects, $\mathcal{U}_{\mathtt{T}}$ is equal to the identity on objects and $\mathcal{U}^{\mathtt{T}} \circ \mathcal{F}^{\mathtt{T}} = \mathtt{T}$. This notion of computations has not given rise to a monadic interpretation in functional programming languages because the existence of the algebraic structure appears as a side condition difficult to encode in the absence of dependent types. To the opposite, the Kleisli category was more suited because it corresponds to the subcategory of free algebras of $\mathcal{C}^{\mathtt{T}}$, and thus the algebraic nature of computational types can be canonically encoded in the interpretation of the arrow.

B. Call-by-name decomposition in CBPV.

The first observation, that was already made in [4], is that the traditional monadic interpretation is call-by-value whereas type theories such as CIC are fundamentally call-by-name. The latter fact comes from the independence on the order of evaluation hardcoded in the conversion rule. We advocate that CBPV allows to clarify the situation by decribing both call-by-value and call-by-name as two distinct decompositions, leading to a more atomic presentation. CBPV's types (and terms) are divided into two classes: pure values and effectful computations. 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 (see Figure 1). Call-by-name and call-by-value strategies can then be decomposed into CBPV, inducing in turn two monadic translations for the simply-typed λ-calculus.

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

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

whereas the by-name decomposition $[-]^{\mathtt{n}}$ is defined as

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

Kleisli adjunction and forcing. When $\mathcal{U}$ and $\mathcal{F}$ are instantiated by the Kleisli adjunction for the reader monad, we recover the forcing translation. It has already been noticed in [4] that the by-value decomposition corresponded to the usual presheaf construction, while the by-name decomposition was the one preserving the conversion rule of type theory.

Eilenberg-Moore adjunction and monadic translations. We now observe that dually, when $\mathcal{U}$ and $\mathcal{F}$ are instantiated by the Eilenberg-Moore adjunction, we recover, in the call-by-value decomposition, the usual monadic interpretation for which

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

Fig. 1. Call-by-push-value (types only)

$A \to B$ is interpreted by $A \to \mathtt{T}\ B$. Quite surprisingly, we recover the definition of an arrow in the Kleisli category whereas we are using the Eilenberg-Moore adjunction. This is simply because the by-value decomposition forces one to consider only the free algebras of the monad, whose category is equivalent to the Kleisli category.

C. By-name Eilenberg-Moore translation for CIC.

The defect of the by-value decomposition comes from the fact that, as CIC is call-by-name, the correctness of the translation requires too many definitional equalities to hold. We advocate in this paper that this is solved by considering by-name decomposition of the Eilenberg-Moore adjunction. In that case, as $\mathcal{U}$ is the identity, the dependent product is translated transparently. Types are translated as plain algebras (*i.e.,* without coherence requirement), which can be easily expressed by the dependent sum

$$\mathbb{\square}_i = \Sigma A : \square_i.\, \mathtt{T}\ A \to A. \tag{1}$$

But in CIC, universes satisfy a kind of self-enrichment expressed as $\square_i : \square_{i+1}$. Thus, to get a correct interpretation of universes, the monad needs to satisfy the additional requirement that the type of algebras needs to be itself an algebra of the monad, that is morally $\mathbb{\square}_i : \mathbb{\square}_{i+1}$. A monad satisfying this property will be called a self-algebraic monad. It is at the heart of the definition of the weaning translation given in the following section.

III. Weaning Translation of CC_ω

This section describes the weaning translation for the negative fragment of CIC, namely CC_ω plus Σ-types, whose typing and computation rules are given in Figure 2. It is based on the notion of *self-algebraic proto-monad.*

Definition 1. A self-algebraic proto-monad is given by the following family of terms:

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

where the universe indices i, j are implicitly universally quantified and $\mathbb{\square}_i$ is defined in (1), furthermore subject to the following definitional equation:

$$\mathtt{El}\ (\mathtt{ret}\ \mathbb{\square}_i\ M) \quad \equiv \quad M.$$

We call this structure a proto-monad because it does not have to satisfy the monadic laws. Only one rule for the interaction between $\mathtt{El}$ and $\mathtt{ret}$ is needed, but at the same time it is crucial that it holds definitionally rather than just

Negative fragment:

$$A, B, M, N ::= \square_i \mid x \mid M\ N \mid \lambda x : A.\, M \mid \Pi x : A.\, B$$

$$\frac{\vdash \Gamma \qquad i < j}{\Gamma \vdash \square_i : \square_j} \qquad \frac{\Gamma \vdash A : \square_i \qquad \Gamma, x : A \vdash B : \square_j}{\Gamma \vdash \Pi x : A.\, B : \square_{\max(i,j)}}$$

$$\frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \Pi x : A.\, B : \square_i}{\Gamma \vdash \lambda x : A.\, M : \Pi x : A.\, B} \qquad \frac{\Gamma \vdash M : \Pi x : A.\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B\{x := N\}} \qquad \frac{\Gamma \vdash M : B \qquad \Gamma \vdash A : \square_i}{\Gamma, x : A \vdash M : B}$$

$$\frac{}{\vdash \cdot} \qquad \frac{\Gamma \vdash A : \square_i}{\vdash \Gamma, x : A} \qquad \frac{\Gamma \vdash A : \square_i}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma \vdash M : B \qquad \Gamma \vdash A : \square_i \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash M : A}$$

$$(\lambda x : A.\, M)\ N \equiv M\{x := N\} \qquad \text{(congruence rules ommitted)}$$

Σ-types:

$$A, B, M, N ::= \ldots \mid \Sigma x : A.\, B \mid M.\pi_1 \mid M.\pi_2 \mid (M, N)$$

$$\frac{\Gamma \vdash A : \square_i \qquad \Gamma, x : A \vdash B : \square_j}{\Gamma \vdash \Sigma x : A.\, B : \square_{\max(i,j)}} \qquad \frac{\Gamma \vdash M : \Sigma x : A.\, B}{\Gamma \vdash M.\pi_1 : A} \qquad \frac{\Gamma \vdash M : \Sigma x : A.\, B}{\Gamma \vdash M.\pi_2 : B\{x := M.\pi_1\}}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B\{x := M\} \qquad \Gamma \vdash \Sigma x : A.\, B : \square_i}{\Gamma \vdash (M, N) : \Sigma x : A.\, B}$$

$$(M, N).\pi_1 \equiv M \qquad (M, N).\pi_2 \equiv N$$

Fig. 2. Typing and computation rules of CC_ω and Σ-types

propositionally. Furthermore, the self-algebraic quality is given by the `El` primitive, which intuitively endows $\square_i$ with a structure of `T`-algebra, even though it is not a formal requirement. Actually, we could make the connection with category theory more precise by asking for propositional equality for the rest of the laws. But propositional equalities cannot be of any use in the syntactical translation because using them would create higher coherence issues, and since the inception of Homotopy Type Theory [7], it is well-known that this turns out to be a subtle matter.

Remark 1. It looks like it is possible to use the fact that $(\texttt{El}\ B).\pi_1$ is always equipped with a weak algebra structure given by $(\texttt{El}\ B).\pi_2$ to canonically derive the heterogeneous binding operation `hbnd` out of `bnd` as

$$\begin{aligned}&\texttt{hbnd} := \lambda A\, B\, M\, F.\\ &\quad (\texttt{El}\ B).\pi_2\ (\texttt{bnd}\ \_\ \_\ M\ (\lambda x : A.\, \texttt{ret}\ \_\ (F\ x)))\end{aligned}$$

but this does not work to interpret inductive datatypes. In general, the resulting term would not satisfy the crucial definitional equation of Definition 3.

Remark 2. For a given monad `T`, there can be several algebra structures on $\square_i$, and thus distinct implementations for `El` as discussed in Section V.

In what follows, we will be using the typical ambiguity of type theory, and we will not explicit the universe indices of the various universe-polymorphic terms. It would be possible to do so at the cost of annotating the type-formers of the source theory, but we consider that the additional burden would not bring more insight into the translation.

$$\begin{aligned}
[\square_i] &:= \texttt{ret}\ \square_{i+1}\ ((\texttt{T}\ \square_i), \mu_{\square_i})\\
[x] &:= x\\
[\lambda x : A.\, M] &:= \lambda x : [\![A]\!].\, [M]\\
[M\ N] &:= [M]\ [N]\\
[\Pi x : A.\, B] &:= \texttt{ret}\ \square\ ((\Pi x : [\![A]\!].\, [\![B]\!]), \mu_{\Pi AB})\\
[\![A]\!] &:= (\texttt{El}\ [A]).\pi_1\\
\mu_{\square_i} &:= \lambda A : \texttt{T}\ (\texttt{T}\ \square_i).\\
&\qquad \texttt{bnd}\ (\texttt{T}\ \square_i)\ \square_i\ A\ (\lambda X : \texttt{T}\ \square_i.\, X)\\
\mu_{\Pi AB} &:= \lambda(\hat{f} : \texttt{T}\ (\Pi x : [\![A]\!].\, [\![B]\!]))\, (x : [\![A]\!]).\\
&\qquad \texttt{hbnd}\ (\Pi x : [\![A]\!].\, [\![B]\!])\ [B]\\
&\qquad\quad \hat{f}\ (\lambda f : \Pi x : [\![A]\!].\, [\![B]\!].\, f\ x)\\
[\![\cdot]\!] &:= \cdot\\
[\![\Gamma, x : A]\!] &:= [\![\Gamma]\!], x : [\![A]\!]
\end{aligned}$$

Fig. 3. Weaning Translation

Definition 2 (Weaning)**.** Assuming a self-algebraic proto-monad, we define the weaning translation from CC_ω to $\mathrm{CC}_\omega + \Sigma$ by induction over the term syntax in Figure 3.

As explain in Section II, the translation is essentially trans-

parent on the functional fragment. Only types are heavily modified. Intuitively, this corresponds to a universe-aware version of the Eilenberg-Moore construction, where all types carry a weak T-algebra structure which does not satisfy any equation. Luckily, on all closed types, the weak algebra structure will definitionally evaluate to a proper algebra.

We now turn to the proof that this translation gives rise to a syntactical model of CC_ω. First, it naturally preserves definitional equalities.

Proposition 1 (Computational soundness)**.** *If $M \equiv N$ then $[M] \equiv [N]$.*

Proof. Congruence rules are transparent owing to the fact the translation is defined by induction on syntax, and the β-rule is transparent as well because any β-redex is translated into another β-redex. □

Remark 3. If the target theory furthermore validates the η-rule for Π-types, it it obvious that the source theory also does, for λ-abstractions and applications are transparently translated. This is another free consequence of the fact that the translation stems from a call-by-push-value decomposition.

Proposition 2 (Type unfolding)**.** *The following definitional equations hold:*

- $[\![\square_i]\!] \equiv \mathtt{T}\ \mathbb{\square}_i$
- $[\![\Pi x : A.\, B]\!] \equiv \Pi x : [\![A]\!].\, [\![B]\!]$

Proof. By application of the reduction rule on $\mathtt{El}$. □

Proposition 3 (Typing soundness)**.** *If $\Gamma \vdash M : A$ then it follows that $[\![\Gamma]\!] \vdash [M] : [\![A]\!]$.*

Proof. The variable, λ-abstraction and application cases are direct, thanks to the application of the conversion rule on Π-types from Proposition 2. Conversion is handled transparently thanks to Proposition 1. The only non-trivial cases are the rules introducing types. All type formers are translated into a term of the form $\mathtt{ret}\ \mathbb{\square}\ (A, \mu_A)$, so by Proposition 2 it is sufficient to check that $(A, \mu_A) : \mathbb{\square} \equiv \Sigma A : \square.\, \mathtt{T}\ A \to A$.

- For the rule $\square_i : \square_{i+1}$, we must therefore show that $(\mathtt{T}\ \mathbb{\square}_i, \mu_{\square_i}) : \mathbb{\square}_{i+1}$. We have indeed $\mathtt{T}\ \mathbb{\square}_i : \square_{i+1}$ because $\mathbb{\square}_i : \square_{i+1}$. It is also obvious that $\mu_{\square_i} : \mathtt{T}\ (\mathtt{T}\ \mathbb{\square}_i) \to \mathtt{T}\ \mathbb{\square}_i$ because this corresponds to the free algebra structure.
- Likewise, for the rule $\Pi x : A.\, B : \square_{\max(i,j)}$, we have $\Pi x : [\![A]\!].\, [\![B]\!] : \square_{\max(i,j)}$ by applying the induction hypothesis. It remains to show that $\mu_{\Pi AB}$ has the expected type, which is a matter of straightforward symbol pushing. □

Thus, we get a model of CC_ω by a translation into CIC.

Theorem 1. *Weaning provides a syntactic model of* CC_ω.

IV. Weaning Translation of BTT

This section extends the weaning translation to inductive types, and in particular to dependent elimination. Because the weaning translation adds effects to the theory, full dependent elimination is not valid anymore. This is a typical issue arising in presence of effects as it has been oberved independently by Barthe and Uustalu [1] and by Herbelin [2]; and which reappeared in our recent work on forcing.

In this section, we present a restriction of dependent elimination that allows for the definition of the weaning translation on inductive types. We call the type theory resulting from this restriction Baclofen Type Theory (BTT), because it alleviates addiction to dependence. It is the source calculus of both the weaning and the forcing translation[2] and, as such, is a good candidate to model effectful type theories.

A. Blaclofen Type Theory

On the negative fragment, BTT coincides with CC_ω. The only difference between BTT and CIC appears on the eliminators of inductive types. While CIC has one notion of pattern-matching that implements both non-dependent and dependent elimination, BTT has two. The first one is non-dependent and is unconditionally valid, while the second one is dependent and restricted through the use of storage operators.

We recall here the definition of storage operators [8], a notion arising from classical realizability. Intuitively, a storage operator on type A is a term $\theta_A : A \to (A \to R) \to R$ allowing to enforce a call-by-value semantics on terms of type A in an otherwise call-by-name language by means of continuation-passing style. When A is an inductive datatype, there is a canonical storage operator on A only defined in terms of non-dependent elimination.

In order to keep the presentation simple, we will not be using the usual fixpoint-based presentation of CIC as a source calculus, but will rather define eliminators. Also, we will only present BTT on three prototypical inductive types: booleans, lists and propositional equality. Those inductive types have been chosen because they feature all the ingredients appearing in generic inductive types: multiple constructors, parameters, indices and recursion. They are defined—in a Coq-like syntax—as:

$$\begin{array}{ll}\mathtt{Inductive}\ \mathtt{bool} : \square := & \mathtt{Inductive}\ \mathtt{list}\ (A : \square) : \square := \\ \mid \mathtt{true} : \mathtt{bool} & \mid \mathtt{nil} : \mathtt{list}\ A \\ \mid \mathtt{false} : \mathtt{bool} & \mid \mathtt{cons} : A \to \mathtt{list}\ A \to \mathtt{list}\ A\end{array}$$

$$\mathtt{Inductive}\ \mathtt{eq}\ (A : \square)\ (x : A) : A \to \square := \mathtt{refl} : \mathtt{eq}\ A\ x\ x$$

In BTT, the non-dependent eliminators are unrestricted and are the same as in CIC. They are given by the following terms, with the usual ι-rules:

$$\begin{aligned}&\mathtt{bool\_case} : \Pi P.\, P \to P \to \mathtt{bool} \to P \\ &\mathtt{list\_case} : \Pi A\, P.\, P \to (A \to \mathtt{list}\ A \to P \to P) \to \\ &\qquad\qquad \mathtt{list}\ A \to P \\ &\mathtt{eq\_case} : \Pi A\, (x : A)\, (P : A \to \square).\, P\ x \to \\ &\qquad\qquad \mathtt{eq}\ A\ x\ y \to P\ y\end{aligned}$$

Storage operators can be defined using only those non-dependent elimination. The storage operators for our prototypical inductive examples are defined in Figure 4. In BTT,

[2] In [4], this restricted version of CIC was referred to as CIC^-.

$$
\begin{array}{lcl}
\theta_{\texttt{bool}} & : & \texttt{bool} \to (\texttt{bool} \to \Box) \to \Box \\
& := & \lambda b\, k.\, \texttt{bool\_case}\ ((\texttt{bool} \to \Box) \to \Box) \\
& & \quad (\lambda k.\, k\ \texttt{true})\ (\lambda k.\, k\ \texttt{false})\ b\ k \\
\theta_{\texttt{list}} & : & \Pi A.\, \texttt{list}\ A \to (\texttt{list}\ A \to \Box) \to \Box \\
& := & \lambda A\, l\, k.\, \texttt{list\_case}\ A\ ((\texttt{list}\ A \to \Box) \to \Box) \\
& & \quad (\lambda k.\, k\ \texttt{nil}) \\
& & \quad\quad (\lambda x\ \_\ r\, k.\, r\ (\lambda l.\, k\ (\texttt{cons}\ A\ x\ l)))\ l\ k \\
\theta_{\texttt{eq}} & : & \Pi A\, (x\, y : A).\, \texttt{eq}\ A\ x\ y \to \\
& & \quad (\Pi y : A.\, \texttt{eq}\ A\ x\ y \to \Box) \to \Box \\
& := & \lambda A\, x\, y\, e\, k.\, \texttt{eq\_case}\ A\ x \\
& & \quad (\lambda y : A.\, (\texttt{eq}\ A\ x\ y \to \Box) \to \Box) \\
& & \quad\quad (\lambda k.\, k\ x\ (\texttt{refl}\ A\ x))\ y\ e
\end{array}
$$

Fig. 4. Storage operators for booleans, lists and equality.

storage operators are used to constraint the evaluation of the predicate on which dependent elimination is performed. On our prototypical inductive examples, dependent eliminators are given by the terms (together with the usual ι-rules):

$$\texttt{bool\_rect} : \Pi P : \texttt{bool} \to \Box.\, P\ \texttt{true} \to P\ \texttt{false} \to \Pi b : \texttt{bool}.\, \theta_{\texttt{bool}}\ b\ P$$

$$\texttt{list\_rect} : \Pi A\, (P : \texttt{list}\ A \to \Box).\, P\ (\texttt{nil}\ A) \to (\Pi x\, l.\, \theta_{\texttt{list}}\ A\ l\ P \to \theta_{\texttt{list}}\ A\ (\texttt{cons}\ A\ x\ l)\ P) \to \Pi l : \texttt{list}\ A.\, \theta_{\texttt{list}}\ A\ l\ P$$

$$\texttt{eq\_rect} : \Pi A\, (x : A)\, (P : \Pi y : A.\, \texttt{eq}\ A\ x\ y \to \Box).\ P\ x\ (\texttt{refl}\ A\ x) \to \Pi y\, e.\, \theta_{\texttt{eq}}\ A\ x\ y\ e\ P$$

Note that the usual dependent elimination of CIC can be decomposed into this restricted elimination followed by an η-rule for inductive types which can be written: $\theta_A\ a\ P = P$ for all inductive types A. While this η-rule is actually propositionally valid in CIC, it is not preserved in presence of effects.

We highlight that BTT is the source theory of both the call-by-name forcing translation and, as we will see, the weaning translation, which itself encompasses a large class of effects. Furthermore, it also prevents the paradox arising from careless mixing of CIC with computational classical logic of Herbelin [2]. Thence we believe that we have enough phenomenological evidence to claim that BTT is the proper way to mix dependent type theory with computational effects.

Conjecture 1. BTT *models effectful type theories.*

Obviously, this claim is somewhat ill-defined. As of today, it is not even clear what an effectful type theory is. As for the Church-Turing thesis, it should rather be taken as a definition of what an effectful type theory is.

B. Dependent Monads

In order to interpret inductive types of BTT, we need to define a stronger notion of proto-monad, which is also equipped with structure allowing to depend on free algebras.

Definition 3. We say that a self-algebraic proto-monad $\texttt{T}$ has dependent elimination whenever the following equation holds

$$\texttt{hbnd}\ A\ B\ (\texttt{ret}\ A\ M)\ F \quad \equiv \quad F\ M$$

and furthermore there exists a universe-polymorphic dependent binding operation

- $\texttt{dbnd}$:
$$\Pi(A : \Box_i)\, (R : [\![\Box_k]\!])\, (B : A \to [\![R]\!] \to \texttt{T}\ \Box_j).\ \Pi(\hat{x} : \texttt{T}\ A)\, (r : [\![R]\!]).\, (\Pi x : A.\, [\![B\ x\ r]\!]) \to (\texttt{El}\ (\texttt{hbnd}\ A\ [R \to \Box_j]\ \hat{x}\ B\ r)).\pi_1$$

subject to the following definitional equation:

$$\texttt{dbnd}\ A\ R\ B\ (\texttt{ret}\ A\ M)\ N\ F \quad \equiv \quad F\ M\ N.$$

For readability purposes, we use the translation macros $[\cdot]$ and $[\![\cdot]\!]$ slightly abusively above, as they do not apply to the target theory, but they can be expanded nevertheless in the same syntactical way, *e.g.,* $[\![B\ x\ r]\!] := (\texttt{El}\ (B\ x\ r)).\pi_1$.

Remark 4. Note that the equation on $\texttt{dbnd}$ is only well-typed whenever the equation on $\texttt{hbnd}$ holds definitionally.

Altough the type of $\texttt{dbnd}$ is intimidating, this is just an artefact due to dependent typing. As witnessed by its reduction rule, this is simply a variant of the $\texttt{hbnd}$ term where B is allowed to depend on the $x : A$ argument, and they share the same computational content. It has additional arguments R and r for technical reasons arising from commutative cuts that are needed to define dependent recursors.

In what follows, we will assume that $\texttt{T}$ is a self-algebraic proto-monad with dependent elimination.

C. Inductive Types without Dependent Elimination

In the next sections, we assume that our target theory implements at least CIC [9], in particular fixpoints with a guard condition.

The translation of inductive datatypes is rather straightforward from a programming point of view. The language we are working in is call-by-name, so that all non-recursive datatypes are interpreted as a free algebra of $\texttt{T}$. For recursive datatypes, recursive calls are turned into the free algebra itself, which raises an issue in CIC. Namely, datatype constructors must satisfy the positivity criterion. In order for our translation to go through, we assume that $\texttt{T}$ is syntactically strictly positive in its argument, which will be the case for all examples from Section V. We define formally the inductive type translation below.

Definition 4. Assume an inductive type $\mathcal{I}$ with parameters $(p_1 : P_1)\ \ldots\ (p_n : P_n)$ and indices $(x_1 : X_1)\ \ldots\ (x_m : X_m)$, with constructors $c_1\ \ldots\ c_k$. We define a new inductive $\mathcal{I}^\bullet$ with parameters $(p_1 : [\![P_1]\!])\ \ldots\ (p_n : [\![P_n]\!])$ and indices $(x_1 : [\![X_1]\!])\ \ldots\ (x_m : [\![X_m]\!])$, with constructors $c_1^\bullet\ \ldots\ c_k^\bullet$ defined as follows. Assume a constructor c of $\mathcal{I}$ of type

$$\Pi(p_1 : P_1) \ldots (p_n : P_n)\, (a_1 : A_1) \ldots (a_l : A_l).\ \mathcal{I}\ p_1\ \ldots\ p_n\ M_1\ \ldots\ M_m$$

with argument types $\vec{A}$, then $c^\bullet$ has type

$$\Pi(p_1 : [\![P_1]\!]) \ldots (p_n : [\![P_n]\!])\, (a_1 : [\![A_1]\!]) \ldots (a_l : [\![A_l]\!]).\ \mathcal{I}^\bullet\ p_1\ \ldots\ p_n\ [M_1]\ \ldots\ [M_m]$$

where we pose locally

$$[\![\mathcal{I}]\!] := \lambda p_1 \ldots p_n\, x_1 \ldots x_m.\, \texttt{T}\ (\mathcal{I}^\bullet\ p_1\ \ldots\ p_n\ x_1\ \ldots\ x_m).$$

$\texttt{Inductive bool}^\bullet : \square :=$
$\mid \texttt{true}^\bullet : \texttt{bool}^\bullet$
$\mid \texttt{false}^\bullet : \texttt{bool}^\bullet$

$\texttt{Inductive list}^\bullet\ (A : \texttt{T}\ \square) : \square :=$
$\mid \texttt{nil}^\bullet : \texttt{list}^\bullet\ A$
$\mid \texttt{cons}^\bullet : [\![A]\!] \to \texttt{T}\ (\texttt{list}^\bullet\ A) \to \texttt{list}^\bullet\ A$

$\texttt{Inductive eq}^\bullet\ (A : \texttt{T}\ \square)\,(x : [\![A]\!]) : [\![A]\!] \to \square :=$
$\mid \texttt{refl}^\bullet : \texttt{eq}^\bullet\ A\ x\ x$

Fig. 5. Examples of inductive types translation

Positivity of c ensures us that only $[\![\mathcal{I}]\!]$ will be needed rather than the more general $[\mathcal{I}]$.

Figure 5 explicits the above translation on the prototypical examples, in a Coq-like syntax. Observe in particular how the the recursive argument of the $\texttt{cons}^\bullet$ constructor is boxed under a monadic type, revealing a typical call-by-name behaviour.

Definition 5 (Translation of inductive types)**.** Given an inductive type $\mathcal{I}$ and its constructors c as before, we extend the $[\cdot]$ translation as follows:

$$\begin{aligned}
[\mathcal{I}] &:= \lambda(\vec{p} : [\![\vec{P}]\!])\,(\vec{x} : [\![\vec{X}]\!]).\,\texttt{ret}\ \square\ ((\texttt{T}\ (\mathcal{I}^\bullet\ \vec{p}\ \vec{x})), \mu_{\mathcal{I}}) \\
[c] &:= \lambda(\vec{p} : [\![\vec{P}]\!])\,(\vec{a} : [\![\vec{A}]\!]).\,\texttt{ret}\ (\mathcal{I}^\bullet\ \vec{p}\ [\vec{M}])\ (c^\bullet\ \vec{p}\ \vec{a}) \\
\mu_{\mathcal{I}} &:= \lambda\hat{\imath} : \texttt{T}\ (\texttt{T}\ (\mathcal{I}^\bullet\ \vec{p}\ \vec{x})).\,\texttt{bnd}\ _\ _\ \hat{\imath}\ (\lambda i.\, i)
\end{aligned}$$

Although the syntax is heavyweight, the translation is essentially pointwise, except that it inserts a few free algebra structures here and there.

Proposition 4. *The translation of inductive types preserves typing.*

Proof. Direct check. In particular, we have the conversion $[\![\mathcal{I}\ \vec{p}\ \vec{x}]\!] \equiv \texttt{T}\ (\mathcal{I}^\bullet\ [\vec{p}]\ [\vec{x}])$. □

We can now define non-dependent eliminators on inductive datatypes by only relying on hbnd. Just as T needed to satisfy a positivity condition, the existence of those eliminators depend on hbnd being syntactically positive as well. We will assume that this is the case in what follows, and once again the examples of Section V will show that this is not a strong restriction in practice.

Rather than giving the generic translation, which would not be very readable, we provide the eliminators on the three prototypical examples.

Definition 6. We extend the translation using these constants for the corresponding eliminators from the source theory, *i.e.,*

$$[\texttt{bool\_case}] := \texttt{bool\_case}^\bullet$$

and similarly for list and eq; the eliminators are defined as:

$\texttt{bool\_case}^\bullet :$
$[\![\Pi P.\, P \to P \to \texttt{bool} \to P]\!]$
$:= \lambda P\, p_t\, p_f\, \hat{b}.\,\texttt{hbnd}\ \_\ \_\ \hat{b}$
$\quad (\lambda b.\,\texttt{match}\ b\ \texttt{with}\ \texttt{true}^\bullet \Rightarrow p_t \mid \texttt{false}^\bullet \Rightarrow p_f)$

$\texttt{list\_case}^\bullet :$
$[\![\Pi A\, P.\, P \to (A \to \texttt{list}\ A \to P \to P) \to \texttt{list}\ A \to P]\!]$
$:= \lambda A\, P\, p_0\, p_S\, \hat{l}.\,\texttt{hbnd}\ \_\ \_\ \hat{l}$
$\quad (\texttt{fix}\ \Phi\ l \Rightarrow \texttt{match}\ l\ \texttt{with}$
$\qquad \mid \texttt{nil}^\bullet\ \_ \Rightarrow p_0$
$\qquad \mid \texttt{cons}^\bullet\ \_\ x\ l \Rightarrow p_S\ x\ l\ (\texttt{hbnd}\ \_\ \_\ l\ \Phi))$

$\texttt{eq\_case}^\bullet :$
$[\![\Pi A\,(x : A)\,(P : A \to \square).\, P\ x \to \texttt{eq}\ A\ x\ y \to P\ y]\!]$
$:= \lambda A\, x\, P\, p\, \hat{e}.\,\texttt{hbnd}\ \_\ \_\ \hat{e}$
$\quad (\lambda e.\,\texttt{match}\ e\ \texttt{with}\ \texttt{refl}^\bullet\ \_\ \_ \Rightarrow p)$

Proposition 5. *The eliminators satisfy the usual ι-rules definitionally.*

Proof. Simple application of the conversion rule of hbnd. □

As one can observe, the eliminator for $\texttt{list}^\bullet$ uses CIC fixpoints in a non-trivial way, and this is precisely from where the requirement that hbnd is syntactically positive stems.

The empty type is an inductive type, so it is straightforward to give a criterion characterizing the consistency of the translated theory.

Proposition 6 (Consistency)**.** *The source theory is consistent iff* $\texttt{T}\ \bot$ *is empty in the target theory.*

Proof. Indeed, $[\![\bot]\!] \equiv \texttt{T}\ \bot^\bullet \cong \texttt{T}\ \bot$ where $\cong$ denotes type isomorphism. □

In Section VII, we will see how to use parametricity techniques to overcome this condition by allowing the inconsistent world to cohabitate with a consistent one even when $\texttt{T}\ \bot$ is inhabited.

D. Dependent Case Analysis

Finally, we show how dependent elimination restricted to storage operators is definable in the source theory, thus leading to a model of BTT.

Proposition 7. *There exists dependent eliminators for* bool, list *and* eq *with the restricted types of Section IV-A which satisfy in addition the usual ι-rules.*

Proof. The dependent eliminators are implemented essentially like their non-dependent counterpart, except that hbnd is turned into dbnd. The additional R and r arguments of dbnd are instantiated respectively with the type of P and P from the corresponding recursor. Validity of ι-rules is in turn ensured by the conversion rule on dbnd. □

Contrarily to the forcing translation, it is sometimes possible to cheat and implement full dependent elimination.

Proposition 8. *If there is a term* $✠ : \Pi A : \square.\,\texttt{T}\ A$ *and one can definitionally discriminate terms of the form* $\texttt{ret}\ A\ M$*, then full dependent elimination is implementable.*

Proof. The eliminator is defined as the standard CIC eliminator on terms which are applied constructors (*i.e.,* of the form $\texttt{ret}\ \_\ (c\ \vec{M})$ for some constructor c) and otherwise uses ✠ to produce a dummy proof. □

Obviously, by Proposition 6 the existence of ✠ entails that the theory resulting from weaning is logically inconsistent, but it may still have some use for programming or proof extraction. Practical examples will be given in the next section.

However, it is shown in Section VII that, using parametricity techniques, it is possible to get effects with full dependent elimination without the cost of inconsistency.

V. Instances of the Weaning Translation

In this section, we describe a few representative instances of our model. The diversity of the effects obtained shows that this construction is quite generic. For the various $\mathtt{T}$ we give, the monad structure is well-known, so that we will only describe the data specific to this article. The corresponding code can be found at `https://github.com/CoqHott/coq-effects`.

Our target theory is computational, thus it is possible to give equational theories for the effectful operators we introduce thereafter. Nonetheless, this section is more about the proof of concept, and hence we will refrain from describing them.

A. Writer on a free monoid

We assume a fixed universe-polymorphic type $\Omega^\bullet$ and we also assume that our target theory features a universe-polymorphic type of lists.

Definition 7. Let $\mathtt{T}\ A := A \times \mathtt{list}\ \Omega^\bullet$ together with

$$\begin{array}{lcl}
\mathtt{El} & := & \lambda A.\, A.\pi_1 \\
\mathtt{hbnd} & := & \lambda A\, B\, \hat{x}\, f.\, \mathtt{match}\ \hat{x}\ \mathtt{with} \\
 & & \quad \mid (x, \mathtt{nil}\ _) \Rightarrow f\ x \\
 & & \quad \mid (x, l) \Rightarrow (\mathtt{El}\ B).\pi_2\ ((f\ x), l) \\
\mathtt{dbnd} & := & \lambda A\, B\, R\, \hat{x}\, r\, f.\, \mathtt{match}\ \hat{x}\ \mathtt{with} \\
 & & \quad \mid (x, \mathtt{nil}\ _) \Rightarrow f\ x\ r \\
 & & \quad \mid (x, l) \Rightarrow (\mathtt{El}\ B).\pi_2\ ((f\ x\ r), l)
\end{array}$$

Note that we are somehow cheating to implement this instance, because we take advantage of the fact we can definitionally observe whether the monoid element is the unit to satisfy the conversion rule on `hbnd`.

Definition 8. We extend the source theory with $\Omega : \square$, $\mathtt{tell} : \Omega \to \mathtt{unit}$ and $\mathtt{listen} : \mathtt{unit} \to \Omega$ as follows where `app` is list concatenation.

$$\begin{array}{lcl}
[\Omega] & := & \mathtt{ret}\ \square\ ((\mathtt{list}\ \Omega^\bullet), \mu_\Omega) \\
\mu_\Omega & := & \lambda\omega.\, \mathtt{app}\ (\omega.\pi_1)\ (\omega.\pi_2) \\
[\mathtt{tell}] & := & \lambda\omega : \mathtt{list}\ \Omega^\bullet.\, ((), \omega) \\
[\mathtt{listen}] & := & \lambda i : \mathtt{unit}^\bullet \times \mathtt{list}\ \Omega^\bullet.\, i.\pi_2
\end{array}$$

It is possible more generally to observe the current written value on universes and inductive datatypes, but we will not describe the relevant combinators here. This translation preserves consistency, but its logical implications are unclear.

B. Dynamic Exceptions

As for the writer, we assume a fixed universe-polymorphic type $\mathtt{E}^\bullet$, and in addition that the target theory has a sum type.

Definition 9. The exception monad is defined as $\mathtt{T}\ A := A + \mathtt{E}^\bullet$ together with

$$\begin{array}{lcl}
\mathtt{El} & := & \lambda A.\, \mathtt{match}\ A\ \mathtt{with}\ \mathrm{in}_l\ X \Rightarrow X \mid \mathrm{in}_r\ e \Rightarrow \mho\ e \\
\mathtt{hbnd} & := & \lambda A\, B\, \hat{x}\, f.\, \mathtt{match}\ \hat{x}\ \mathtt{with} \\
 & & \quad \mid \mathrm{in}_l\ x \Rightarrow f\ x \\
 & & \quad \mid \mathrm{in}_r\ e \Rightarrow (\mathtt{El}\ B).\pi_2\ (\mathrm{in}_r\ e) \\
\mathtt{dbnd} & := & \lambda A\, B\, R\, \hat{x}\, r\, f.\, \mathtt{match}\ \hat{x}\ \mathtt{with} \\
 & & \quad \mid \mathrm{in}_l\ x \Rightarrow f\ x\ r \\
 & & \quad \mid \mathrm{in}_r\ e \Rightarrow (\mho\ e).\pi_2\ (\mathrm{in}_r\ e)
\end{array}$$

where $\mho : \mathtt{E}^\bullet \to \square$ is arbitrary, *e.g.,* $\mho := \lambda\_.\, (1, \lambda\_.\, ())$.

The fact that $\mho$ is arbitrary reflects the choice we have to make about types returning exceptions, because it is not specified at all. It fixes the meaning of $\Gamma \vdash M : \mathtt{fail}\ N\ \square$, where `fail` is defined below.

Definition 10. We extend the source theory with $\mathtt{E} : \square$ and $\mathtt{fail} : \mathtt{E} \to \Pi A : \square.\, A$ defined as follows.

$$\begin{array}{lcl}
[\mathtt{E}] & := & \mathtt{ret}\ \square\ (\mathtt{E}^\bullet, \mu_\mathtt{E}) \\
\mu_\mathtt{E} & := & \lambda e.\, \mathtt{match}\ e\ \mathtt{with}\ \mathrm{in}_l\ e \Rightarrow e \mid \mathrm{in}_r\ e \Rightarrow e \\
[\mathtt{fail}] & := & \lambda e\, A.\, (\mathtt{El}\ A).\pi_2\ (\mathrm{in}_r\ e)
\end{array}$$

The reader should be aware that the exceptions arising from this translation are call-by-name, so that they do not behave like their usual call-by-value counterpart. In particular, we have $[\mathtt{fail}\ M\ (\Pi x : A.\, B)] \equiv [\lambda x : A.\, \mathtt{fail}\ M\ B]$. This means that exceptions cannot be caught on Π-types. We can catch them on universes and inductive types though, because they are translated as free algebras. For instance, there exists in the source theory a term $\mathtt{catch}_\mathtt{bool}$ of type

$$\begin{array}{l}
\Pi P : \mathtt{bool} \to \square.\, P\ \mathtt{true} \to P\ \mathtt{false} \to \\
\quad (\Pi e : \mathtt{E}.\, P\ (\mathtt{fail}\ e\ \mathtt{bool})) \to \Pi b : \mathtt{bool}.\, P\ b
\end{array}$$

with the expected reduction rules on all three cases.

Because of Proposition 6, the resulting theory is inconsistent as soon as $\mathtt{E}^\bullet$ is inhabited. Yet, the translation can be used for logical purposes. The careful reader may indeed have realized that the weaning translation on exceptions is no more than Friedman's trick [10] on steroids. Thus we can use it for program extraction on the Σ^0_1 classical fragment.

Proposition 9. *We have* $[\![\neg\neg A]\!] \cong ([\![A]\!] \to \mathtt{E}^\bullet) \to \mathtt{E}^\bullet$.

Proposition 10. *If $\mathcal{I}$ is a first-order type, then there is a term* $\mathtt{eval}_\mathcal{I} : [\![\mathcal{I}]\!] \to \mathcal{I} + \mathtt{E}^\bullet$.

By Proposition 8, full dependent elimination is interpreted, so that by putting all these properties together we recover this interesting extraction result by taking $\mathtt{E}^\bullet := \mathcal{I}$.

Theorem 2. *If $\mathcal{I}$ is a first-order type and there is a proof of $\neg\neg\mathcal{I}$ in* CIC*, then there is a proof of $\mathcal{I}$ in* CIC.

C. Non-determinism

We sketch a model of BTT with non-determinism here.

Definition 11. The non-empty list monad is defined as $\mathtt{T}\ A := A \times \mathtt{list}\ A$ together with

$$\begin{array}{lcl}
\mathtt{El}\ (X, (\mathtt{nil}\ _)) & := & X \\
\mathtt{El}\ (X, [X_1; \ldots; X_n]) & := & (X.\pi_1 \times X_1.\pi_1 \ldots \times X_n.\pi_1, \mu)
\end{array}$$

where μ is canonically defined from the algebra structure on the X_i. Likewise, it is possible to define `hbnd` and `dbnd` similarly to the usual `bnd` combinator.

Note that just as in the case of exceptions, `El` could be defined in many other ways, because effectful types are not specified. For instance, it is also valid to take $\texttt{El}\ A := A.\pi_1$.

In any case, this monad allows to write a merging operator that features non-determinism.

Definition 12. We extend the source theory with an operator $\texttt{amb} : \Pi A : \square.\ A \to A \to A$ defined as follows.

$$[\texttt{amb}] \quad := \quad \lambda A\, x\, y.\, (\texttt{El}\ A).\pi_2\ (x, (\texttt{cons}\ \_\ y\ (\texttt{nil}\ \_)))$$

Despite its type, `amb` is neither the first nor the second projection. On closed types, it is possible to give an equational theory to `amb` (*e.g.,* it is associative). Once again, the model preserves consistency, but its added logical expressivity is not known.

D. Non-termination

It turns out that the delay monad is indeed a self-algebraic monad, so that it is possible to build a type theory with built-in non-termination and arbitrary fixpoints.

Definition 13. The delay monad is defined as

$$\begin{array}{l}\texttt{CoInductive}\ \texttt{T}\ (A : \square) : \square := \\ \mid \texttt{here} : A \to \texttt{T}\ A \\ \mid \texttt{next} : \texttt{T}\ A \to \texttt{T}\ A\end{array}$$

with for instance $\texttt{El}\ (\texttt{here}\ \_\ X) := X$ and $(\texttt{unit}, \lambda\_.\,())$ otherwise. It has dependent elimination in a direct way.

The definition of `El` above is very naive, and implies that all impure types evaluate to the singleton type. This is degenerated, and it is possible to do better by using a heterogeneous delay monad instead.

Following Capretta's work [11], we can define a fixpoint operator through the weaning translation.

Proposition 11. *Assuming B is a free algebra, there exists a term $\texttt{Y}_B : [\![\Pi A.\, ((A \to B) \to A \to B) \to A \to B]\!]$ that satisfies a fixpoint equation (up to bisimilarity).*

Proof. The free algebra condition on B means indeed that we have $[\![B]\!] := \texttt{T}\ C$ for some C, so that we must build $\texttt{Y}_B : \Pi A.\, ((([\![A]\!] \to \texttt{T}\ C) \to [\![A]\!] \to \texttt{T}\ C) \to [\![A]\!] \to \texttt{T}\ C$. It is then defined by coinduction in the same way as in Capretta's paper. □

Note that any type can be turned into a free algebra by using the `box` type from the next section so that one can define a generic fixpoint. The resulting theory is inconsistent by virtue of Proposition 6, but this does not prevent its use as a dependently-typed programming language with decidable typing *and* general recursion.

VI. Linearity as a Guard Condition

The restriction on dependent elimination of BTT can be extended on a more semantical ground by using linearity, a notion that has been first described by Munch-Maccagnoni [12] and rephrased recently in the context of CBPV by Levy [13]. Linearity[3] is a property of call-by-name functions which intuitively represents the fact that a function is semantically call-by-value. We recall Levy's definition below in a type-theoretical setting, which first relies on the existence of a unary sum type.

Definition 14. We define the inductive type `box` as

$$\texttt{Inductive}\ \texttt{box}\ (A : \square) : \square := \texttt{Box} : A \to \texttt{box}\ A$$

together with its non-dependent eliminator

$$\texttt{box\_case} : \Pi (A : \square)\, (P : \square).\, (A \to P) \to \texttt{box}\ A \to P$$

which both automatically admit a weaning translation.

Naturally, in CIC, `box` A is isomorphic to A, thanks to dependent elimination. This is not the case anymore in BTT, precisely because full dependent elimination is not available. Instead, the fact that this type is an inductive type provides a way to locally perform computation in a call-by-value fashion. This is made clear through the weaning translation, as witnessed by the following lemma.

Proposition 12. $[\![\texttt{box}\ A]\!] \cong \texttt{T}\ [\![A]\!]$ *holds in* CIC.

We can now define the proper notion of linearity in the weaning translation.

Definition 15 (Linearity)**.** We say that a term $M : [\![A \to B]\!]$ is *linear* whenever there is a proof of

$$\Pi \hat{x} : [\![\texttt{box}\ A]\!].\, [\lambda f.\, f\ (\texttt{box\_case}\ A\ A\ (\lambda x : A.\, x)\ \hat{x})]\ M = [\lambda f.\, \texttt{box\_case}\ A\ B\ f\ \hat{x}]\ M$$

in the target theory.

This definition uses equality of the target theory, so that we can define a predicate $\texttt{linear} : \Pi A\, B.\, [\![A \to B]\!] \to \square$ in the target theory. Although it is defined here in terms of the weaning translation, it can be generalized to any syntactic translation of type theory in a straightforward way. In particular, in a pure language, the notion is trivial. For instance, in CIC, by taking $[\cdot]$ to be the identity, all functions are linear because `box` A is isomorphic to A.

This is not true in presence of non-trivial effects. Let for instance be $\varphi : \texttt{unit} \to \texttt{unit} := \lambda\_.\,()$, and assume that the language allows non-termination. Thus there is a non-converging term $\texttt{loop} : \texttt{box unit}$. Then the linearity equation is not valid for $M := \varphi$ and $\hat{x} := \texttt{loop}$, because the left-hand side converges, while the right-hand side diverges.

At first sight, it seems that linearity is a property that makes sense in the context of semantics of programming languages, but which has little to do with logic. Surprisingly, it turns out to

[3]This notion of linearity is mostly unrelated to linearity in the sense of linear use of variables.

be a valuable criterion to assess that a predicate is insensitive to storing.

Proposition 13 (Storing invariance)**.** *Let us assume that* `T` *satisfies the monad laws propositionally, that* `hbnd` *is extensionally defined as in Remark 1, and furthemore that the target theory satisfies functional extensionality. Let $\mathcal{I}$ be a non-recursive inductive type.*

If a predicate $P : [\![\mathcal{I} \to \Box]\!]$ is linear, then

$$\Pi i : [\![\mathcal{I}]\!].\, [\theta_{\mathcal{I}}]\ i\ P = P\ i.$$

Proof. As $\theta_{\mathcal{I}}$ is defined as a single pattern-matching, it unfolds to an application of `hbnd` to a case analysis. But $[\Box]$ is a free algebra, and by the canonicity of `hbnd`, this can be rewritten to an application of `bnd` to a case analysis. Then, using the isomorphism of Proposition 12, the equation of linearity is transported on `T` $\mathcal{I}^\bullet$. We conclude by a rewriting of monadic laws and dependent elimination on $\mathcal{I}^\bullet$. □

The above lemma uses type theory in a non-trivial way, because to be formulated it requires that predicates are actual terms from the theory, and thus relies on the existence of universes. We suspect this is the reason why the few attempts at an effectful dependent type theory have been strongly restricted up to now.

Confortingly, this property is not limited to the weaning translation as it holds also for call-by-name forcing under the same assumption. This is quite expected, as the definitions of BTT and linearity do not dependent on the underlying effect. Furthermore, it is also obviously true for the identity translation of CIC.

Remark 5. Storing invariance does not hold for recursive inductive types. This is due to the fact that linearity on f only captures the fact that f commutes with one pattern-matching, whereas it would need to commute with arbitrary fixpoints to extend to recursive types.

Notwithstanding the above limitation, storing invariance allows to provide a semantical restriction on non-recursive dependent elimination in BTT rather than having to go through storage operators. Namely, it is semantically correct to provide a dependent pattern-matching in BTT on, say, `bool` of the following form.

$$\frac{\ldots \qquad \text{(usual rules)} \qquad \ldots \qquad C \text{ linear in } z}{\Gamma \vdash \texttt{match}\ M\ \texttt{with true} \Rightarrow N_1 \mid \texttt{false} \Rightarrow N_2 : C\{z := M\}}$$

The key difference with standard CIC pattern-matching is the side condition of linearity on C.

Obviously, it is not palatable to have to write proofs of linearity, and one would like to have the system decide automatically whether a predicate is linear. Just as the fixpoint guard condition of CIC is a syntactic underapproximation of a semantical notion of positivity, it is possible to provide a syntactic approximation for linearity. A few syntactical closure properties are described in Levy's paper for instance. Most notably, storage operators would syntactically turn an arbitrary predicate into a linear one, because they start with a linear pattern-matching on the variable being eliminated. It means that there would not be any need to hardwire them in BTT.

Apart from the fact it is not clear how to extend linearity to recursive inductive types, this would result in a pattern-matching-based theory closer to CIC, and easier to work with in a proof assistant. We will refrain from exploring this further in this paper and leave it for future work.

VII. Parametric Weaning Translation for CIC

This section is dedicated to the description of a technique based on parametricity to recover a syntactical model of full CIC out of the weaning translation, while allowing at the same time this pure world to cohabitate with effectful programs.

Parametricity is a well-known proof technique introduced by Reynolds [14] that is used for a vast range of purposes, *e.g.,* the automatic derivation of properties of programs according to their type described by Wadler [15]. A few years ago, Bernardy and others showed that parametricity could be written as a syntactical translation over type theory [16]. In a nutshell, their translation associates to a term $M : A$ a term $[M]_\varepsilon : [A]_\varepsilon\ M$ where $[A]_\varepsilon$ translates to the parametricity predicate over A. The elegance of the translation stems from the fact that just as for CC_ω, it does not make any formal distinction between terms and types and that the parametricity predicate $[A]_\varepsilon$ and parametricity theorem $[M]_\varepsilon$ are actually derived by the same syntactical induction.

Contrarily to their presentation, we will not use parametricity to state properties on terms from an underlying theory but rather use it to carve out non-parametric terms from the underlying theory, resulting in a syntactical model of CIC instead of BTT.

Moreover, the translation we give is a variant of parametricity that we will call *internal*, by opposition to Bernardy's presentation which we will call *external*. The difference lays in the handling of free variables. In Bernardy's original presentation, terms from the source theory cannot depend on the fact that variables are assumed parametric. This is due to terms being translated in two sequents, where the first translation is essentially identity for unary parametricity, and the second one explicits parametricity of the first term.

By contrast, we will call our translation internal because it only produces one sequent, and furthermore translated terms rely essentially on the fact that parametricity assumptions on variables are available. In particular, as a syntactical enhancement, variables from the context will be translated as Σ-types rather than as a pair of variables.

A. Parametricity Restriction

The intuitive idea of this model is simple. As the weaning translation fails to interpret dependent elimination because there are non-canonical inductive terms in the model, let us simply claim that we will only be interested in inductive terms which are well-behaved, *i.e.,* parametric. Obviously this needs to scale to the whole translation, so that it needs to be a little involved. We first explicit the translation on the negative

$$
\begin{array}{lcl}
[\square_i] & := & \texttt{ret}\ \square_{i+1}\ ((\texttt{T}\ \square_i), \mu_{\square_i}) \\
[x] & := & x.\pi_1 \\
[\lambda x : A.\, M] & := & \lambda x : [\![A]\!]^!.\, [M] \\
[M\ N] & := & [M]\ [N]^! \\
[\Pi x : A.\, B] & := & \texttt{ret}\ \square\ ((\Pi x : [\![A]\!]^!.\, [\![B]\!]), \mu_{\Pi AB}) \\
[\square_i]_\varepsilon & := & \lambda A : \texttt{T}\ \square_i.\, (\texttt{El}\ A).\pi_1 \to \square_i \\
[x]_\varepsilon & := & x.\pi_2 \\
[\lambda x : A.\, M]_\varepsilon & := & \lambda x : [\![A]\!]^!.\, [M]_\varepsilon \\
[M\ N]_\varepsilon & := & [M]_\varepsilon\ [N]^! \\
[\Pi x : A.\, B]_\varepsilon & := & \lambda f : \Pi x : [\![A]\!]^!.\, [\![B]\!]. \\
 & & \quad \Pi x : [\![A]\!]^!.\, [B]_\varepsilon\ (f\ x) \\
\mu_{\square_i} & := & \lambda A : \texttt{T}\ (\texttt{T}\ \square_i). \\
 & & \quad \texttt{bnd}\ (\texttt{T}\ \square_i)\ \square_i\ A\ (\lambda X : \texttt{T}\ \square_i.\, X) \\
\mu_{\Pi AB} & := & \lambda(\hat{f} : \texttt{T}\ (\Pi x : [\![A]\!]^!.\, [\![B]\!]))\, (x : [\![A]\!]^!). \\
 & & \quad \texttt{hbnd}\ (\Pi x : [\![A]\!]^!.\, [\![B]\!])\ [B] \\
 & & \quad\quad \hat{f}\ (\lambda f : \Pi x : [\![A]\!]^!.\, [\![B]\!].\, f\ x) \\
[\![A]\!] & := & (\texttt{El}\ [A]).\pi_1 \\
[M]^! & := & ([M], [M]_\varepsilon) \\
[\![A]\!]^! & := & \Sigma x : [\![A]\!].\, [A]_\varepsilon\ x \\
[\![\cdot]\!] & := & \cdot \\
[\![\Gamma, x : A]\!] & := & [\![\Gamma]\!], x : [\![A]\!]^!
\end{array}
$$

Fig. 6. Parametric Weaning Translation

fragment before showing how this allows to recover full-blown dependent elimination.

Definition 16 (Parametric Weaning)**.** Assuming $\texttt{T}$ is a self-algebraic proto-monad, we define parametric weaning over the syntax of CC_ω in Figure 6.

On the $[\cdot]$ fragment, the translation is essentially the same as the non-parametric weaning, except that variables and function arguments are packed with their parametricity proof. This is typical of the thunking appearing in call-by-name translations, and is reminiscent of the forcing translation [4], which is why we use the same $[\cdot]^!$ notation.

In what follows, for the sake of simplicity we will assume that the target theory validates definitional surjective pairing. It is technically not necessary, at a cost of a slightly less understandable translation, where $[\cdot]^!$ is primitive and $[\cdot]$ and $[\cdot]_\varepsilon$ are derived from it.

Proposition 14 (Substitution lemma)**.** *We have both* $[M\{x := N\}] \equiv [M]\{x := [N]^!\}$ *and* $[M\{x := N\}]_\varepsilon \equiv [M]_\varepsilon\{x := [N]^!\}$.

Proof. By induction over M. □

Corollary 1 (Computational soundness)**.** *If* $M \equiv N$ *then* $[M] \equiv [N]$ *and* $[M]_\varepsilon \equiv [N]_\varepsilon$.

Proposition 15 (Typing soundness)**.** *If* $\Gamma \vdash M : A$ *then we have* $[\![\Gamma]\!]^! \vdash [M] : [\![A]\!]$ *and* $[\![\Gamma]\!]^! \vdash [M]_\varepsilon : [A]_\varepsilon\ [M]$*. As a consequence,* $[\![\Gamma]\!]^! \vdash [M]^! : [\![A]\!]^!$.

Proof. By induction over the typing derivation. □

Remark 6. To extend the theory with some constant $M : A$, it is thus sufficient to provide a pair of terms $[M] : [\![A]\!]$ and $[M]_\varepsilon : [A]_\varepsilon\ [M]$ as all other definitions are derived.

As before, the previous lemmas state no more that the $[\cdot]^!$ translation gives rise to a syntactic model of CC_ω. It is slightly more convoluted that the bare weaning translation, as it adds a predicate to types and ensure that all terms respect the predicate of their type.

Proposition 16. *We have the following unfoldings.*

- $[\![\square]\!]^! \equiv \Sigma A : \texttt{T}\ \square.\, (\texttt{El}\ A).\pi_1 \to \square$
- $$
  \begin{array}{rcl}
  [\![\Pi x : A.\, B]\!]^! & \equiv & \Sigma f : \Pi x : [\![A]\!]^!.\, [\![B]\!]. \\
  & & \qquad \Pi x : [\![A]\!]^!.\, [B]_\varepsilon\ (f\ x) \\
  & \cong & \Pi x : [\![A]\!]^!.\, [\![B]\!]^!
  \end{array}
  $$

We now show how it actually extends to inductive types. We will focus on the simple example of booleans, because it already exhibits the trick, although the same technique scales to arbitrary inductive types.

Definition 17 (Boolean parametricity)**.** We define the following inductive predicate over $\texttt{T}\ \texttt{bool}^\bullet$.

$$
\begin{array}{l}
\texttt{Inductive}\ \texttt{bool}^\bullet_\varepsilon : \texttt{T}\ \texttt{bool}^\bullet \to \square := \\
\mid \texttt{true}^\bullet_\varepsilon : \texttt{bool}^\bullet_\varepsilon\ (\texttt{ret}\ \texttt{bool}^\bullet\ \texttt{true}^\bullet) \\
\mid \texttt{false}^\bullet_\varepsilon : \texttt{bool}^\bullet_\varepsilon\ (\texttt{ret}\ \texttt{bool}^\bullet\ \texttt{false}^\bullet)
\end{array}
$$

Definition 18. We extend the parametric weaning translation as follows.

$$
\begin{array}{lcl}
[\texttt{bool}] & := & \texttt{ret}\ \square\ ((\texttt{T}\ \texttt{bool}^\bullet), \mu_{\texttt{bool}}) \\
[\texttt{true}] & := & \texttt{ret}\ \texttt{bool}^\bullet\ \texttt{true}^\bullet \\
[\texttt{false}] & := & \texttt{ret}\ \texttt{bool}^\bullet\ \texttt{false}^\bullet \\
[\texttt{bool}]_\varepsilon & := & \texttt{bool}^\bullet_\varepsilon \\
[\texttt{true}]_\varepsilon & := & \texttt{true}^\bullet_\varepsilon \\
[\texttt{false}]_\varepsilon & := & \texttt{false}^\bullet_\varepsilon \\
\mu_{\texttt{bool}} & := & \lambda \hat{b} : \texttt{T}\ (\texttt{T}\ \texttt{bool}^\bullet).\, \texttt{bnd}\ _\ _\ \hat{b}\ (\lambda b.\, b)
\end{array}
$$

It is not hard to check that typing soundness still holds with this extension.

Proposition 17. *The above translation preserves typing.*

Luckily, through the translation, boxed inhabitants have more structure than just pertaining to the free algebra over $\texttt{bool}^\bullet$.

Proposition 18. *We have* $[\![\texttt{bool}]\!]^! \equiv \Sigma b : \texttt{T}\ \texttt{bool}^\bullet.\, \texttt{bool}^\bullet_\varepsilon\ b$.

This proposition amounts to say that $[\![\texttt{bool}]\!]^!$ is isomorphic to $\texttt{bool}$ simply thanks to the second component. This allows to write full dependent elimination over booleans.

Proposition 19. *There exists a fully dependent eliminator over* `bool` *in the parametric weaning translation, that furthermore preserves usual ι-rules.*

Proof. It is implemented by dependent pattern-matching over the second component of $b : [\![\texttt{bool}]\!]^!$. Reduction rules are valid owing to the fact they apply to constructors. □

It is essential for the eliminator to have access to the parametricity proof of its argument. This is where the *internal* nature of our variant stands out. With external parametricity, the eliminator would only have access to a non-standard boolean $b : \texttt{T bool}^\bullet$ and would not be able to do anything relevant with it without the storage operator restriction. Even though the parametricity proof of the eliminator would access it, it would already be too late.

This technique can be generalized effortlessly to any inductive type in a way similar to Bernardy's parametricity, leading to the following observation.

Theorem 3. *The parametric weaning translation is a syntactic model of* CIC. *In addition, it preserves consistency.*

Interestingly enough, the parametric part of the translation is quite independent from the effectful translation. It can actually be extended to other settings, such as call-by-name forcing. Moreover, it highlights a blind spot of parametricity: *types*.

The parametricity restriction enforces functions to be respectful, and inductive datatypes to be indeed inductively generated. Yet, the only constraint we put on types is that they wear a parametricity predicate on their elements. But the weaning translation *did* generate new types as soon as `T` is not the identity, as $[\![\Box]\!] \equiv \texttt{T}\,\Box$. Thus there are new anomalous types modelled by the parametric weaning translation, so that it is larger than the original theory. Clearly, it is impossible in pure CIC to add a constraint on the shape of a type precisely because of parametricity. But here, we have more structure on types so that we could restrict them. For instance, we could enforce elements of $\texttt{T}\,\Box$ to be of the form $\texttt{ret}\,\Box\,A$ for some A to bar non-standard types and recover the original theory.

The usual parametricity translation simply does not put any constraint. We believe that this explains the mechanism hidden under the monotonicity condition of forcing, which preserves inductive types while allowing to expand universes as presheaves. Somehow, universes in type theory are *underspecified*.

B. BTT *as a Modality*

It is obvious that giving a model of CIC alone is not very exciting. The interesting bits come from the additional expressive power bestowed by this translation. In this section, we show that BTT can be injected into the parametric weaning translation thanks to the effect modality.

Definition 19 (Effect modality)**.** We define the *effect modality* $\mathbb{E} : \Box_i \to \Box_i$ in the source theory as follows.

$$\begin{array}{lcl} [\mathbb{E}] & := & \lambda A : [\![\Box]\!]^!.\, A.\pi_1 \\ [\mathbb{E}]_\varepsilon & := & \lambda (A : [\![\Box]\!]^!)\,(x : (\texttt{El}\ (A.\pi_1)).\pi_1).\,\texttt{unit} \end{array}$$

The proper way to understand the effect modality is that given a type A with the parametricity predicate A_ε, it builds the same underlying type A but with the full predicate, so that any term of type $[\![A]\!]$ is parametric for $[\mathbb{E}\ A]_\varepsilon$.

Example 1. For instance, we have

$$[\![\mathbb{E}\ \texttt{bool}]\!]^! \equiv \Sigma b : \texttt{T bool}^\bullet.\,\texttt{unit} \cong \texttt{T bool}^\bullet.$$

Proposition 20. *The effect modality has a few properties in the source theory that should be highlighted.*

- *It is definitionally idempotent,* i.e., $\mathbb{E}\ (\mathbb{E}\ A) \equiv \mathbb{E}\ A$.
- *There exists a return operator* $\eta_{\mathbb{E}} : \Pi A.\, A \to \mathbb{E}\ A$.
- *It is* not *functorial, as* $\mathbb{E}\ (\Pi x : A.\, B) \cong \Pi x : A.\, \mathbb{E}\ B$.

In particular, $\mathbb{E}$ is not a self-algebraic monad. Nonetheless, it is possible to give an internal translation of BTT into the parametric weaning of CIC.

Definition 20 (Effectful translation)**.** The effectful translation $[\cdot]_{\mathbb{E}}$ is defined as $[\cdot]$ in Figure 6 except that we replace uniformly all translations by their $\mathbb{E}$-indexed variant, where

$$\begin{array}{lcl} [M]^!_{\mathbb{E}} & := & ([M]_{\mathbb{E}}, ()) \\ [\![A]\!]_{\mathbb{E}} & := & (\texttt{El}\ [A]_{\mathbb{E}}).\pi_1 \\ [\![A]\!]^!_{\mathbb{E}} & := & \Sigma x : [\![A]\!]_{\mathbb{E}}.\,\texttt{unit} \end{array}$$

Note in particular that $[\cdot]_\varepsilon$ does not appear in $[\cdot]_{\mathbb{E}}$.

Proposition 21. *Assuming* $\Gamma \vdash M : A$ *in* BTT*, then we have* $[\![\Gamma]\!]^!_{\mathbb{E}} \vdash [M]^!_{\mathbb{E}} : [\![A]\!]^!_{\mathbb{E}}$.

Proof. Essentially the same proof as in Proposition 3. □

Definition 21 (Effectful quotation)**.** For every term M from BTT, we extend the source theory with a term $\lceil M \rceil$ whose translation is defined below.

$$[\lceil M \rceil] \ := \ [M]_{\mathbb{E}} \qquad\qquad [\lceil M \rceil]_\varepsilon \ := \ ()$$

Definition 22. We define in the source theory the effectful element $\mathcal{El} : \mathbb{E}\ \Box_i \to \Box_i$ as follows.

$$\begin{array}{lcl} [\mathcal{El}] & := & \lambda A : [\![\mathbb{E}\ \Box]\!]^!.\, A.\pi_1 \\ [\mathcal{El}]_\varepsilon & := & \lambda (A : [\![\mathbb{E}\ \Box]\!]^!)\,(x : (\texttt{El}\ (A.\pi_1)).\pi_1).\,\texttt{unit} \end{array}$$

Proposition 22. *We have* $\mathbb{E}\ (\mathcal{El}\ A) \equiv \mathcal{El}\ A$.

The following theorem is then an immediate consequence of Proposition 21 obtained by unfolding the definitions.

Theorem 4. *Assuming* $\Gamma \vdash M : A$ *in* BTT*, then we have* $[\![\mathcal{El}\ \lceil \Gamma \rceil]\!]^! \vdash [\lceil M \rceil]^! : [\![\mathcal{El}\ \lceil A \rceil]\!]^!$ *in the target theory.*

This shows that we can readily embed BTT into CIC as a kind of DSL. Obviously, the $\lceil \cdot \rceil$ operator satisfies a few rules that are not described here, but which follow from the definitions, *e.g.,* $\lceil \Pi x : A.\, B \rceil \equiv \eta_{\mathbb{E}}\ \Box\ (\Pi x : \mathcal{El}\ \lceil A \rceil.\, \mathcal{El}\ \lceil B \rceil)$. The major difference of this translation w.r.t. the direct inclusion of BTT into CIC as one would have obtained by Proposition 3 comes from the fact universes are translated transparently. Hence, we can reason about the effectful fragment as if it was part of the theory, and this can be used to mix effects with proofs in a consistent way.

VIII. Related Work

The introduction of dependent types in mainstream programming languages is currently gaining traction, as witnessed by the growing community of Idris [17] for instance. This naturally raises the question of computational effects in a dependent setting. However, the literature on the topic is rather terse. To the best of our knowledge, apart from the work on forcing in type theory [4], there are only two very recent and overlapping lines of work that intersect with BTT, namely the independent description of a dependent CBPV by Ahman et al. [18] and Vákár [19].

Quite interestingly, the initial intuition given by Ahman et al. for their system is very similar to the fundamental idea of weaning that types must carry an algebra structure and that dependency is distributed over it. But, in the content of the paper, their presentation turns into a more ad-hoc system where dependency-introducing structures are duplicated between terms and computations, resulting in a syntax cancelling all the fundamental computational dualities of CBPV.

As for the systems described by Vákár, their limitations are clear through the prism of BTT. Vákár describes two systems, dCBPV^- where dependence is restricted to values, and dCBPV^+, which is an extension of the former. dCBPV^- is compatible with BTT on the CBN fragment, but its limitation lies in its lack of expressivity. There are no universes and no dependent elimination on anything else than syntactical values, so that it is not even clear how to introduce a non-trivial conversion rule. dCBPV^+ is therefore proposed to extend dCBPV^- with the so-called dependent Kleisli extension rule. It turns out that this rule essentially postulates in the logic that all predicates are linear, which is, as we have seen in Section VI, false in presence of effects. As a consequence, all models of dCBPV^+ are either pure or logically unsound. All the examples he gives of the implementation of the Kleisli extension rule are actually an instance of Proposition 8.

On another line of work, the notion of (accessible) modalities in Homotopy Type Theory [7] is quite close to self-algebraic monads. Indeed, the purpose of modalities is to reflect a subuniverse of the theory with extra logical properties, in the same way as self-algebraic monads are used to define new models of BTT with additional effects. However, such modalities are by definition idempotent, whereas self-algebraic monads, coming from functional programming, rarely are.

IX. Conclusion and Future Work

In this paper, we have defined the first monadic translation of type theory, called weaning translation, that allows for a large range of effects in dependent type theory—such as exceptions, non-termination, non-determinism or writing operation. The weaning translation typically applies to a version of CIC with a restricted version of dependent elimination, called Baclofen Type Theory, which constitutes a good candidate for the definition of an effectful type theory. Finally, we have shown how to recover a translation of full CIC using parametricity techniques. The translation is available as a Coq plugin and all the examples have been formalized with it.

Our work opens a new line of research on the logical expressivity induced by the use of traditional computation monads in type theory. It also offers the possibility to program with and reason on effects in the same framework, without relying on additional axioms in the theory.

References

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

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

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

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

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

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

[7] T. Univalent Foundations Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*, Institute for Advanced Study, 2013.

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

[9] C. Paulin-Mohring, "Introduction to the Calculus of Inductive Constructions," in *All about Proofs, Proofs for All*, ser. Studies in Logic (Mathematical logic and foundations), B. W. Paleo and D. Delahaye, Eds., Jan. 2015, vol. 55.

[10] H. Friedman, "Classically and intuitionistically provably recursive functions," in *Higher Set Theory*, ser. Lecture Notes in Mathematics, G. H. Müller and D. S. Scott, Eds. Springer Berlin Heidelberg, 1978, vol. 669, pp. 21–27.

[11] V. Capretta, "General recursion via coinductive types," *Logical Methods in Computer Science*, vol. 1, no. 2, 2005.

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

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

[14] J. C. Reynolds, "Types, abstraction and parametric polymorphism," in *IFIP Congress*, 1983, pp. 513–523.

[15] P. Wadler, "Theorems for free!" in *Functional Programming Languages and Computer Architecture*. ACM Press, 1989, pp. 347–359.

[16] J.-P. Bernardy and M. Lasson, "Realizability and Parametricity in Pure Type Systems," in *Foundations of Software Science and Computational Structures*, vol. 6604, Saarbrücken, Germany, Mar. 2011, pp. 108–122.

[17] E. Brady, "Idris, a general-purpose dependently typed programming language: Design and implementation," *Journal of Functional Programming*, vol. 23, no. 05, pp. 552–593, 2013.

[18] D. Ahman, N. Ghani, and G. D. Plotkin, "Dependent types and fibred computational effects," in *19th International Conference on Foundations of Software Science and Computation Structures*. Eindhoven, The Netherlands: Springer Berlin Heidelberg, 2016, pp. 36–54.

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