The Definitional Side of the Forcing

Guilhem Jaber
Gabriel Lewertowski
IRIF - Université Paris Diderot
πr^2 - Inria

Pierre-Marie Pédrot
Inria

Matthieu Sozeau
IRIF - Université Paris Diderot
πr^2 - Inria

Nicolas Tabareau
Inria

Abstract

This paper studies forcing translations of proofs in dependent type theory, through the Curry-Howard correspondence. Based on a call-by-push-value decomposition, we synthesize two simply-typed translations: i) one call-by-value, corresponding to the translation derived from the presheaf construction as studied in a previous paper; ii) one call-by-name, whose intuitions already appear in Krivine and Miquel's work. Focusing on the call-by-name translation, we adapt it to the dependent case and prove that it is compatible with the definitional equality of our system, thus avoiding coherence problems. This allows us to use any category as forcing conditions, which is out of reach with the call-by-value translation. Our construction also exploits the notion of storage operators in order to interpret dependent elimination for inductive types. This is a novel example of a dependent theory with side-effects, clarifying how dependent elimination for inductive types must be restricted in a non-pure setting. Being implemented as a Coq plugin, this work gives the possibility to formalize easily consistency results, for instance the consistency of the negation of Voevodsky's univalence axiom.

Categories and Subject Descriptors F.4.1 [*MATHEMATICAL LOGIC AND FORMAL LANGUAGES*]: Mathematical Logic

Keywords Forcing, Dependent type theory, Inductive types, Effects, Coq

1. Introduction

Forcing has been introduced by Cohen to prove the independence of the Continuum Hypothesis in set theory. The main idea is to build, from a model M, a new model M' for which validity is controlled by a partially-ordered set (poset) of forcing conditions living in M. Technically, a forcing relation $p \Vdash \phi$ between a forcing condition p and a formula ϕ is defined, such that ϕ is true in M' iff $p \Vdash \phi$ is true in M, for some p approximating the new elements of M'. Categorical ideas have been used by Lawvere and Tierney [14] to recast forcing in terms of topos of (pre)sheaves. It is then straightforward to extend the construction to work on categories of forcing conditions, rather than simply posets, giving a proof relevant version of forcing.

Recent years have seen a renewal of interest for forcing, driven by Krivine's classical realizability [9]. In this line of work, forcing is studied as a proof translation, and one seeks to understand its computational content [3, 12], through the Curry-Howard correspondence. This means that $p \Vdash \phi$ is studied as a syntactic translation of formulas, parametrized by a forcing condition p.

Following these ideas, a forcing translation has been defined in [6] for the Calculus of Constructions, the type theory behind the Coq proof assistant. It is based heavily on the presheaf construction of Lawvere and Tierney. The main goal of [6] was to extend the logic behind Coq with new principles, while keeping its fundamental properties: soundness, canonicity and decidability of type checking. This approach can be seen, following [1], as type-theoretic metaprogramming.

However, this technique suffers from coherence problems, which complicate greatly the translation. More precisely, the translation of two definitionally equal terms are not in general definitionally equal, but only propositionally equal. Rewriting terms must then be inserted inside the definition of the translation. If this is possible to perform, albeit tedious, when the forcing conditions form a poset, it becomes intractable when we want to define a forcing translation parametrized by a category of forcing conditions.

In this paper, we propose a novel forcing translation for the Calculus of Constructions (CC_ω), which avoids these coherence problems. Departing from the categorical intuitions of the presheaf construction, it takes its roots in a call-by-push-value [10] decomposition of our system. This will justify to name our translation *call-by-name*, while the previous translation of [6] is *call-by-value*.

"Call-by-name forcing provides the first effectful translation of CC_ω into itself which preserves definitional equality."

We then extend our translation to inductive types by exploiting storage operators [8]—an old idea of Krivine to simulate call-by-value in call-by-name in the context of classical realizability—to restrict the power of dependent elimination in presence of effects. The necessity of a restriction should not be surprising and was already present in Herbelin's work [5].

This provides *the first version of Calculus of Inductive Constructions (*CIC*) with effects*. The nice property of preservation of

LICS '16, July 05-08, 2016, New York, NY, USA

DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2933575.2935320

definitional equality is emphasized by the implementation of a Coq plugin[1] which works for any term of CIC.

We conclude the paper by using forcing to produce various results around homotopy type theory. First, we prove that (a simple version of) functional extensionality is preserved in any forcing layer. Then we show that the negation of Voevodsky's univalence axiom is consistent with CIC plus functional extensionality. This statement could already be deduced for the existence of a set-based *proof-irrelevant* model [16], but we provide the first formalization of it, in a proof relevant setting, and by an easy use of the forcing plugin. Finally, we show that under an additional assumption of monotonicity of types, we get the preservation of (a simple version of) the univalence axiom.

Plan of the paper. In Section 2, we derive, in a simply typed setting, our call-by-name forcing translation using a *call-by-push value decomposition* of the language. We then define the translation for CC_ω, a language with dependent product (§ 3). Its soundness relies on some equality holding *definitionally*, that we get using a Yoneda construction (§ 4). The translation is then extended to datatypes (§ 5), introducing a restriction on CIC to handle *dependent elimination*. The generalization to recursive types is studied in Section 6, relying on *storage operators* to deal with their dependent eliminations. Finally, in Section 7, we use forcing to prove that the negation of *univalence* is consistent with CIC, and discuss a refined translation which enforces some *naturality* conditions, so that univalence is preserved by the translation.

2. Call-by-Push-Value

In this section, we explain how the call-by-push-value language (CBPV) of Levy [10] can be used to present two versions of the forcing translation. To keep our presentation as simple as possible, we will only use a small subset of it, although most of the results can be adapted to a more general setting. The idea of CBPV is to break up the simply-typed λ-calculus, leading to a more atomic presentation distinguishing values and computations, and allowing to add effects easily into the language. We use it as the source language for a generic forcing translation thought of as adding side-effects. Call-by-name and call-by-value strategies can then be decomposed into CBPV, inducing in turn two forcing translations for the λ-calculus.

2.1 Syntax of CBPV

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. The syntax and typing rules are given at Figure 1

We give some intuition behind those terms. The `thunk` primitive is to be understood as a way of *boxing* a computation into a value. Its dual `force` *runs* the computation. Note that this name has nothing to do with forcing itself and is a coincidence. The `return` primitive creates a pure computation from a value. The `let` binding first evaluates its argument, possibly generating some effects, binds the purified result to the variable and continues with the remaining term. Intuitively, this language is no more than the usual decomposition of a monad into an adjunction.

For technical reasons, we endow CBPV with reduction rules that are weaker than what is usually assumed, by restricting substitution to strong values, i.e. values which are not variables, while the standard reduction allows substitution for any value. Indeed, the forcing translation which we present after only interprets this restricted reduction.

[1] Available at `https://github.com/CoqHott/coq-forcing`.

Definition 1 (Restricted CBPV reduction)**.** Strong values $\tilde{v}$ are simply defined as $\tilde{v} := \mathtt{thunk}\ t$. We define the restricted CBPV reduction as the congruence closure of the following generators.

$$\begin{array}{lcl} (\lambda x : A.\, t)\ \tilde{v} & \to & t\{x := \tilde{v}\} \\ \mathtt{let}\ x : A := \mathtt{return}\ \tilde{v}\ \mathtt{in}\ t & \to & t\{x := \tilde{v}\} \\ \mathtt{force}\ (\mathtt{thunk}\ t) & \to & t \end{array}$$

We write $\equiv$ for the equivalence generated by this reduction.

2.2 Simply-Typed Decompositions

We recall here the decompositions of the simply-typed λ-calculus into CBPV. They were actually the original motivation for the introduction of CBPV itself. We will translate the usual λ-calculus where types are described by the inductive grammar

$$A, B := \alpha \mid A \to B$$

using the standard syntax. The results of this section are well-known so we will not dwell on them.

Definition 2. The by-name reduction of the λ-calculus is the congruence closure of the generator

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

while the restricted by-value reduction is the congruence closure of the generator

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

where v is a λ-abstraction.

Definition 3 (By-value decomposition)**.** The by-value decomposition is defined as follows.

$$\begin{array}{lcl} [\alpha]^{\mathtt{v}} & := & \alpha \\ [A \to B]^{\mathtt{v}} & := & \mathcal{U}\,([A]^{\mathtt{v}} \to \mathcal{F}\,[B]^{\mathtt{v}}) \\ [x]^{\mathtt{v}} & := & \mathtt{return}\ x \\ [t\ u]^{\mathtt{v}} & := & \mathtt{let}\ f := [t]^{\mathtt{v}}\ \mathtt{in} \\ & & \mathtt{let}\ x := [u]^{\mathtt{v}}\ \mathtt{in}\ \mathtt{force}\ f\ x \\ [\lambda x : A.\, t]^{\mathtt{v}} & := & \mathtt{return}\ (\mathtt{thunk}\ (\lambda x : [A]^{\mathtt{v}}.\, [t]^{\mathtt{v}})) \end{array}$$

Proposition 1. *If* $\Gamma \vdash t : A$ *then* $[\Gamma]^{\mathtt{v}} \vdash_c [t]^{\mathtt{v}} : \mathcal{F}\,[A]^{\mathtt{v}}$.

Proposition 2. *If* $t \to_{\mathtt{v}} u$ *then* $[t]^{\mathtt{v}} \equiv [u]^{\mathtt{v}}$.

Definition 4 (By-name decomposition)**.** The by-name decomposition is defined as follows.

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

Proposition 3. *If* $\Gamma \vdash t : A$ *then* $\mathcal{U}\,[\Gamma]^{\mathtt{n}} \vdash_c [t]^{\mathtt{n}} : [A]^{\mathtt{n}}$.

Proposition 4. *If* $t \to_{\mathtt{n}} u$ *then* $[t]^{\mathtt{n}} \equiv [u]^{\mathtt{n}}$.

2.3 Forcing Translation

We now define the forcing translation from CBPV into a small dependent extension of the simply-typed λ-calculus. Dependency is needed because we have to be able to state in the type that some relation holds between two elements. For simplicity, we can use for instance the much richer system defined at Section 3. We use implicit arguments and infix notation for clarity when the typing is clear from context.

First of all, we need a notion of preorder in the target calculus.

Definition 5 (Preorder)**.** A preorder is given by

$$
\begin{array}{ll}
\text{value types} & A, B ::= \mathcal{U}\, X \mid \alpha \\
\text{computation types} & X, Y ::= A \to X \mid \mathcal{F}\, A \\
\text{environments} & \Gamma ::= \cdot \mid \Gamma, x : A \\
\\
\text{value terms} & v ::= x \mid \mathtt{thunk}\ t \\
\text{computation terms} & t, u ::= \lambda x : A.\, t \mid t\ v \mid \mathtt{let}\ x : A := t\ \mathtt{in}\ u \mid \mathtt{force}\ t \mid \mathtt{return}\ v
\end{array}
$$

$$
\frac{(x : A) \in \Gamma}{\Gamma \vdash_v x : A} \qquad \frac{\Gamma \vdash_c t : X}{\Gamma \vdash_v \mathtt{thunk}\ t : \mathcal{U}\, X} \qquad \frac{\Gamma \vdash_v v : \mathcal{U}\, X}{\Gamma \vdash_c \mathtt{force}\ v : X} \qquad \frac{\Gamma, x : A \vdash_c t : X}{\Gamma \vdash_c \lambda x : A.\, t : A \to X} \qquad \frac{\Gamma \vdash_v v : A}{\Gamma \vdash_c \mathtt{return}\ v : \mathcal{F}\, A}
$$

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

Figure 1. Call-by-push-value

- a type $\mathbb{P}$;
- a binary relation $\leq$;
- a term $\mathtt{id} : \Pi p : \mathbb{P}.\, p \leq p$;
- a term $\circ : \Pi(p\, q\, r : \mathbb{P}).\, p \leq q \to q \leq r \to p \leq r$

subject to the following conversion rules.

$$\mathtt{id}_p \circ f \equiv f \quad f \circ \mathtt{id}_q \equiv f \quad f \circ (g \circ h) \equiv (f \circ g) \circ h$$

We assume in the remainder of this section a fixed preorder that we will call *forcing conditions*.

Definition 6 (Ground types)**.** We assume given for every CBPV ground type α:

- a type α_p in the target calculus for each $p : \mathbb{P}$;
- a lifting morphism $\theta_\alpha : \Pi(p\, q : \mathbb{P}).\, p \leq q \to \alpha_p \to \alpha_q$

subject to the following conversion rules.

$$\theta_\alpha\ \mathtt{id}_p\ x \equiv x \quad \theta_\alpha\ (f \circ g)\ x \equiv \theta_\alpha\ g\ (\theta_\alpha\ f\ x)$$

Definition 7 (Type translation)**.** The forcing translation on types associates to every CBPV type and forcing condition a target type defined inductively as follows.

$$
\begin{array}{lcl}
[\![\alpha]\!]_p & := & \alpha_p \\
[\![\mathcal{U}\, X]\!]_p & := & \Pi q : \mathbb{P}.\, p \leq q \to [\![X]\!]_q \\
[\![A \to X]\!]_p & := & [\![A]\!]_p \to [\![X]\!]_p \\
[\![\mathcal{F}\, A]\!]_p & := & [\![A]\!]_p
\end{array}
$$

Proposition 5 (Value lifting)**.** *The lifting morphisms of Definition 6 can be generalized to any value type A as θ_A with the same distribution rules.*

Proof. By induction on A. Our only non-variable value type is $\mathcal{U}\, X$ where $\theta_{\mathcal{U}\, X}$ is defined by precomposition. □

Definition 8 (Term translation)**.** The term translation is indexed by an CBPV environment Γ and a preorder variable p and produces a term in the target calculus. It is defined inductively as

$$
\begin{array}{lcl}
[x]_p^\Gamma & := & x \\
[\mathtt{thunk}\ t]_p^\Gamma & := & \lambda(q : \mathbb{P})\, (f : p \leq q).\, \theta_\Gamma\ (f, [t]_q^\Gamma) \\
[\mathtt{force}\ v]_p^\Gamma & := & [v]_p^\Gamma\ p\ \mathtt{id}_p \\
[\lambda x : A.\, t]_p^\Gamma & := & \lambda x : [\![A]\!]_p.\, [t]_p^{\Gamma, x:A} \\
[t\ v]_p^\Gamma & := & [t]_p^\Gamma\ [v]_p^\Gamma \\
[\mathtt{let}\ x : A := t\ \mathtt{in}\ u]_p^\Gamma & := & (\lambda x : [\![A]\!]_p.\, [u]_p^{\Gamma, x:A})\ [t]_p^\Gamma \\
[\mathtt{return}\ v]_p^\Gamma & := & [v]_p^\Gamma
\end{array}
$$

where the $\theta_\Gamma\ (f, t)$ notation stands for $t\{\vec{x} := \theta_{\vec{A}}\ f\ \vec{x}\}$ for each $(x_i : A_i) \in \Gamma$.

The only non-trivial case of this translation is the `thunk` case, which requires to lift all the free variables of the considered term. We need to do this because the resulting term is *boxed* w.r.t. the current forcing condition by a λ-abstraction, so that there is a mismatch between the free variables of $[t]_q^\Gamma$ 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.

Proposition 6 (Typing soundness)**.** *Assume $\Gamma \vdash_c t : X$, then $p : \mathbb{P}, [\![\Gamma]\!]_p \vdash [t]_p^\Gamma : [\![X]\!]_p$ and similarly for values.*

Proposition 7 (Computational soundness)**.** *For all $\Gamma \vdash_c t,u : A$, if $t \equiv u$ then $[t]_p^\Gamma \equiv [u]_p^\Gamma$ and similarly for values.*

The interest of giving this translation directly in CBPV is that we can recover two translations of the λ-calculus by composing it with the by-name and by-value decompositions. This provides hints about the source of the technical impediments encountered in [6].

To start with, we can easily observe that $[\![[A \to B]^{\mathtt{v}}]\!]_p$ is equal to $\Pi q : \mathbb{P}.\, p \leq q \to [\![A]\!]_q \to [\![B]\!]_q$, which is indeed the usual way to translate the arrow type in forcing, as in [6]. The term translation is also essentially the same, except for the adaptations to the dependently-typed case. The two following defects of the call-by-value forcing translation are then obvious through this decomposition.

First, the translation only preserves call-by-value reduction, and not unrestricted β-reduction. Indeed, through the by-value decomposition, a redex translation $[(\lambda x : A.\, t)\ u]^{\mathtt{v}}$ is convertible with $\mathtt{let}\ x := [u]^{\mathtt{v}}\ \mathtt{in}\ [t]^{\mathtt{v}}$, which is itself convertible with $[t]^{\mathtt{v}}\{x := [u]^{\mathtt{v}}\}$ only when $[u]^{\mathtt{v}}$ is a value. Therefore, the interpretation of the conversion rule of CIC by a plain conversion is not possible. One has to resort to more semantical arguments, implying the use of explicit rewriting in the terms.

Second, the very computational conditions imposed over θ_α are highly problematic as soon as we have second-order quantifications. Indeed, we need to ship with each abstracted type $\Pi\alpha : \square.\, A$ a corresponding θ_α in the translation. But then we loose the definitional equalities required by Definition 6. The only thing we can do is to enforce them by using propositional equalities, which will imply in turn some explicit rewriting.

Meanwhile, the by-name variant is way more convenient to use to interpret CIC conversion. Indeed, it interprets the whole β-conversion, and furthermore it does not even require any θ_α for abstracted variables. This is because all value types appearing in

the $[-]^{\mathrm{n}}$ decomposition are of the form $\mathcal{U}\,X$ for some X, so that we statically know we will only need the $\theta_{\mathcal{U}X}$ function which is defined regardless of X. Both properties make a perfect fit for an interpretation of CIC.

3. Forcing Translation in the Negative Fragment

In this section, we first consider the forcing translation of CC_ω, a type theory featuring only negative connectives, i.e. Π-types. It features a denumerable hierarchy of universes $\Box_i$ together with an impredicative universe $*$, and is therefore essentially Luo's ECC without pairs nor cumulativity [11].

This translation builds upon the call-by-name forcing described in the previous section. The main differences are that we handle higher-order and dependency, as well as a presentation artifact where we delay the whole-term lifting of the `thunk` translation by using forcing contexts instead. Moreover, we now consider *categories* of forcing conditions, rather than posets.

Definition 9 (Typing system)**.** As usual, we define here two statements mutually recursively. The statement $\vdash \Gamma$ means that the environment Γ is well-founded, while $\Gamma \vdash M : A$ means that the term M has type A in environment Γ. We write $\Box$ for $*$ or $\Box_i$ for some $i \in \mathbb{N}$. The typing rules are given at Figure 2.

Definition 10 (Forcing context)**.** Forcing contexts σ are given by the following inductive grammar.

$$\sigma ::= p \mid \sigma \cdot x \mid \sigma \cdot (q, f)$$

A forcing context σ may be seen as a path from the initial condition p to a current condition q. The forcing context $\sigma \cdot (q, f)$ extends the path σ upto the new condition q through the path f between p and q.

In the above definition, p, x, q and f are variables binding in the right of the forcing context, and therefore forcing contexts obey the usual freshness conditions obtained through α-equivalence.

We will often write $\sigma \cdot \varphi$ to represent the forcing context σ extended with some forcing suffix φ made of any kind of extension.

Definition 11 (Forcing context validity)**.** A forcing context σ is valid in a context Γ, written $\Gamma \vdash \sigma$, whenever they pertain to the following inductive relation.

$$\frac{}{\cdot \vdash p} \qquad \frac{\Gamma \vdash \sigma}{\Gamma \vdash \sigma \cdot (q, f)} \qquad \frac{\Gamma \vdash \sigma}{\Gamma, x : A \vdash \sigma \cdot x}$$

Definition 12 (Category)**.** A category is given by:

- A term $\vdash \mathbb{P} : \Box_0$ representing objects;
- A term $\vdash \mathtt{Hom} : \mathbb{P} \to \mathbb{P} \to \Box_0$ representing morphisms;
- A term $\vdash \mathtt{id} : \Pi p : \mathbb{P}.\, \mathtt{Hom}\ p\ p$ representing identity;
- A term $\vdash \circ : \Pi(p\,q\,r : \mathbb{P}).\, \mathtt{Hom}\ p\ q \to \mathtt{Hom}\ q\ r \to \mathtt{Hom}\ p\ r$ representing composition.

For readability purposes, we write $\mathtt{id}_p$ for $\mathtt{id}\ p$, $\mathtt{Hom}(p, q)$ for $\mathtt{Hom}\ p\ q$ and we consider the objects for the composition as implicit and write $f \circ g$ for $\circ\ p\ q\ r\ f\ g$ for some objects p, q and r.

Furthermore, we require that we have the following definitional equalities.

$$\frac{\Gamma \vdash f : \mathtt{Hom}(p, q)}{\Gamma \vdash \mathtt{id}_p \circ f \equiv f} \qquad \frac{\Gamma \vdash f : \mathtt{Hom}(p, q)}{\Gamma \vdash f \circ \mathtt{id}_q \equiv f}$$

$$\frac{\Gamma \vdash f : \mathtt{Hom}(p, q) \quad \Gamma \vdash g : \mathtt{Hom}(q, r) \quad \Gamma \vdash h : \mathtt{Hom}(r, s)}{\Gamma \vdash f \circ (g \circ h) \equiv (f \circ g) \circ h}$$

Note that asking that they are given definitionally rather than as mere propositional equalities is, as we will see in Section 4, actually not restrictive.

Definition 13. The last condition σ_e from a forcing context σ is a variable defined inductively as follows.

$$p_e := p \quad (\sigma \cdot x)_e := \sigma_e \quad (\sigma \cdot (q, f))_e := q$$

The morphism of a variable x in a forcing context σ, written $\sigma(x)$, is a term defined inductively as follows.

$$p(x) := \mathtt{id}_p \qquad (\sigma \cdot x)(x) := \mathtt{id}_{\sigma_e}$$
$$(\sigma \cdot y)(x) := \sigma(x) \quad (\sigma \cdot (q, f))(x) := \sigma(x) \circ f$$

Notation 1. As it is a recurring pattern in the translation, we will use the following macros.

$$\lambda(q\,f : \sigma).\, M \quad := \quad \lambda(q : \mathbb{P})\,(f : \mathtt{Hom}(\sigma_e, q)).\, M$$
$$\Pi(q\,f : \sigma).\, M \quad := \quad \Pi(q : \mathbb{P})\,(f : \mathtt{Hom}(\sigma_e, q)).\, M$$

Definition 14 (Forcing translation)**.** The forcing translation is inductively defined on terms as follows.

$$\begin{aligned}
[*]_\sigma &:= \lambda(q\,f : \sigma).\, \Pi(r\,g : \sigma \cdot (q, f)).\, * \\
[\Box_i]_\sigma &:= \lambda(q\,f : \sigma).\, \Pi(r\,g : \sigma \cdot (q, f)).\, \Box_i \\
[x]_\sigma &:= x\ \sigma_e\ \sigma(x) \\
[\lambda x : A.\, M]_\sigma &:= \lambda x : [\![A]\!]^!_\sigma.\, [M]_{\sigma \cdot x} \\
[M\ N]_\sigma &:= [M]_\sigma\ [N]^!_\sigma \\
[\Pi x : A.\, B]_\sigma &:= \lambda(q\,f : \sigma).\, \Pi x : [\![A]\!]^!_{\sigma \cdot (q,f)}.\, [\![B]\!]_{\sigma \cdot (q,f) \cdot x} \\
[\![A]\!]_\sigma &:= [A]_\sigma\ \sigma_e\ \mathtt{id}_{\sigma_e} \\
[M]^!_\sigma &:= \lambda(q\,f : \sigma).\, [M]_{\sigma \cdot (q,f)} \\
[\![A]\!]^!_\sigma &:= \Pi(q\,f : \sigma).\, [\![A]\!]_{\sigma \cdot (q,f)}
\end{aligned}$$

Note that the three last definitions are simple macros definable in terms of the basic forcing translation that will be used pervasively to ease the reading. In particular, the $[-]^!_\sigma$ and $[\![-]\!]^!_\sigma$ macros correspond respectively to the interpretation of `thunk` and $\mathcal{U}$ in the call-by-push-value decomposition.

Assuming that $\Gamma \vdash \sigma$, which we will do implicitly afterwards, we now define the forcing translation on contexts as follows.

$$\begin{aligned}
[\![\cdot]\!]_p &:= p : \mathbb{P} \\
[\![\Gamma]\!]_{\sigma \cdot (q,f)} &:= [\![\Gamma]\!]_\sigma, q : \mathbb{P}, f : \mathtt{Hom}(\sigma_e, q) \\
[\![\Gamma, x : A]\!]_{\sigma \cdot x} &:= [\![\Gamma]\!]_\sigma, x : [\![A]\!]^!_\sigma
\end{aligned}$$

We now turn to the proof that this translation indeed preserves the typing rules of our theory. As proper typing rules and conversion rules are intermingled, we should actually prove it in a mutually recursive fashion, but this would be fairly unreadable. Therefore, in the following proofs, we rather assume that computational (resp. typing) soundness are already proved for the induction hypotheses, in an open recursion style. This is a mere presentation artifact: the loop is tied at the end by plugging the two soundness theorems together.

Proposition 8 (Condition Concatenation)**.** *For any $\Gamma \vdash M : A$, and forcing contexts σ, φ, ψ with φ containing only conditions and morphisms,*

$$[\![\Gamma]\!]_{\sigma \cdot \varphi \cdot \psi} \vdash [M]_{\sigma \cdot (q,f) \cdot \psi}\{q := (\sigma \cdot \varphi)_e, f := (\varphi)\} \equiv [M]_{\sigma \cdot \varphi \cdot \psi}$$

where (φ) stands for the composition of all morphisms in φ.

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

$$\frac{\vdash \Gamma \qquad i < j}{\Gamma \vdash \Box_i : \Box_j} \qquad \frac{\vdash \Gamma}{\Gamma \vdash * : \Box_i} \qquad \frac{\Gamma \vdash A : \Box_i \qquad \Gamma, x : A \vdash B : \Box_j}{\Gamma \vdash \Pi x : A.\, B : \Box_{\max(i,j)}} \qquad \frac{\Gamma \vdash A : \Box_i \qquad \Gamma, x : A \vdash B : *}{\Gamma \vdash \Pi x : A.\, B : *}$$

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

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

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

Figure 2. Typing rules of CC_ω

Proof. By induction over M. This property relies heavily on the fact that the categorical equalities are definitional, and the proof actually amounts to transporting those equalities. □

Proposition 9 (Substitution Lemma)**.** *For any* $\Gamma \vdash M : A$*,*

$$[\![\Gamma]\!]_{\sigma \cdot \varphi} \vdash [M\{x := P\}]_{\sigma \cdot \varphi} \equiv [M]_{\sigma \cdot x \cdot \varphi}\{x := [P]^!_\sigma\}$$

Proof. By induction over M and application of the previous lemma. □

Theorem 1 (Computational Soundness)**.** *If* $\Gamma \vdash M \equiv N$ *then* $[\![\Gamma]\!]_\sigma \vdash [M]_\sigma \equiv [N]_\sigma$.

Proof. The congruence rules are obtained trivially, owing to the fact that the translation is defined by induction on the terms. The β-reduction step is obtained by a direct application of the substitution lemma, while the η-expansion rule is interpreted as-is in the translation. □

Theorem 2 (Typing Soundness)**.** *The following holds.*

- *If* $\vdash \Gamma$ *then* $\vdash [\![\Gamma]\!]_\sigma$.
- *If* $\Gamma \vdash M : A$ *then* $[\![\Gamma]\!]_\sigma \vdash [M]_\sigma : [\![A]\!]_\sigma$.

Proof. By induction on the typing derivation. The only non-immediate case is the conversion rule which is obtained by applying the computational soundness theorem. □

Forcing Layer. We now explain how to use the forcing translation to extend safely CIC with new logical principles, so that type-checking remains decidable and the resulting extended theory is equiconsistent with Coq (i.e. if the empty type of Coq is not inhabited, then neither is the empty type of the resulting theory) as soon as the type $\mathbb{P}$ of objects is inhabited.

In the *forcing layer*, it is possible to add new symbols to the system. Each symbol $\varphi : \Phi$ has to come with its translation $\vdash \varphi^\bullet : \Pi p : \mathbb{P}.\, [\![\Phi]\!]_p$ in CIC. This is done in the Coq plugin using the command

$$\texttt{Forcing Definition}\ \varphi : \Phi\ \texttt{using}\ \mathbb{P}\ \texttt{Hom}.$$

where $\mathbb{P}$ and `Hom` define the category of forcing conditions in use. Note the similarity with forcing in set theory, where a new model is obtained by adding a generic element G to a ground model, and the forcing relation describes inside the ground model the properties of G in the new model.

The typing relation $\vdash_{\mathcal{F}}$ in the layer is defined by extending CIC with the axiom $\Gamma \vdash_{\mathcal{F}} \varphi : \Phi$. By posing $[\varphi]_\sigma := \varphi^\bullet\ \sigma_e$, it is easy to derive that if $\Gamma \vdash_{\mathcal{F}} M : A$ then $[\![\Gamma]\!]_\sigma \vdash [M]_\sigma : [\![A]\!]_\sigma$ using Theorem 2. The abovementioned equiconsistency result is just a consequence of the fact that if $\Gamma \vdash_{\mathcal{F}} M : \bot$ then $[\![\Gamma]\!]_p \vdash [M]_p : \Pi(q\ f : p).\, \bot$, which shows that a proof of the empty type $\bot$ in the forcing layer directly gives a proof of $\bot$ in CIC as soon as the type $\mathbb{P}$ of objects is inhabited.

4. Yoneda to the Rescue

A key property in the preservation of typing is that the forcing category implements category laws in a definitional way. This may seem a very strong requirement. Indeed, any non-trivial operation is going to block on variable arguments, which puts the convertibility at stake. For instance, simply taking objects to be the unit type and morphisms to be booleans equipped with `xor` already breaks at least one of the two identity rules, depending on the order in which `xor` is defined.

Luckily, we can rely on a folklore trick to build for any category an equivalent category with laws that holds definitionally. The soundness of the translation is no more than the good old Yoneda lemma.

Definition 15 (Yoneda translation)**.** Assume a category as given in Definition 12 without assuming any equalities. We define the Yoneda translation of this category as follows.

$$\begin{array}{lcl} \mathbb{P}_{\mathcal{Y}} & := & \mathbb{P} \\ \texttt{Hom}_{\mathcal{Y}}\ p\ q & := & \Pi r : \mathbb{P}.\, \texttt{Hom}(q, r) \to \texttt{Hom}(p, r) \\ \texttt{id}_{\mathcal{Y}}\ p & := & \lambda(r : \mathbb{P})\ (k : \texttt{Hom}(p, r)).\, k \\ \circ_{\mathcal{Y}}\ p\ q\ r\ f\ g & := & \lambda(s : \mathbb{P})\ (k : \texttt{Hom}(r, s)).\, g\ s\ (f\ s\ k) \end{array}$$

Proposition 10 (Yoneda lemma)**.** *The Yoneda translation of a category is a category with laws that holds definitionally.*

Proof. Immediate. Typing is straightforward and equalities are simple $\beta\eta$-equivalences. □

The interesting subtlety of this proof is that we actually do not even need the categorical laws of the base category to recover definitional equalities in the Yoneda translation. What we have done amounts to building the free category generated by objects and morphisms, and definitional equalities follow just because the meta-theory (here, our type theory) is computational. Although the relation between the Yoneda lemma, CPS and free categories has already been observed in the literature, we believe that our current usecase is novel.

It remains now to prove that the Yoneda category is equivalent to its base category. As there is no widely accepted notion of being equivalent in type theory, we are going to allow ourselves to cheat a little bit.

Definition 16 (Equivalence functors)**.** We define two type-theoretic functors $\mathcal{Y}$ (resp. $\mathcal{Y}^{-1}$) between a base category and its Yoneda translation (resp. the converse) as follows. On objects, the translation is the identity

$$\mathcal{Y}_o := \lambda p : \mathbb{P}.\, p \qquad \mathcal{Y}^{-1}_o := \lambda p : \mathbb{P}_{\mathcal{Y}}.\, p$$

while on morphisms we pose

$$\begin{array}{lcl} \mathcal{Y}_h\ p\ q\ f : \mathtt{Hom}_{\mathcal{Y}}\ p\ q & := & \lambda(r : \mathbb{P})\,(k : \mathtt{Hom}(q, r)).\, f \circ k \\ \mathcal{Y}^{-1}_h\ p\ q\ f : \mathtt{Hom}\ p\ q & := & f\ q\ \mathtt{id}_q \end{array}$$

We need to reason about equality, so we suppose until the end of this section that our target type theory features a propositional equality $=$ as defined usually, and furthermore that the functional extensionality principle is provable.

Proposition 11 (Functoriality)**.** *Assuming that equalities of Definition 12 hold propositionally, the above objects are indeed functors, i.e. they obey the usual commutation rules w.r.t. identity and composition propositionally.*

Proposition 12 (Category equivalence)**.** *The above functors form an equivalence in the following sense.*

1. *Assuming that equalities of Definition 12 hold propositionally, then $\mathcal{Y}^{-1}_h\ p\ q\ (\mathcal{Y}_h\ p\ q\ f) = f$ propositionally.*
2. *Assuming parametricity over the quantification on the base category, then $\mathcal{Y}_h\ p\ q\ (\mathcal{Y}^{-1}_h\ p\ q\ f) = f$ propositionally.*

Proof. The first equality is straightforward. The second one is essentially an unfolding of the definition of parametricity over the categorical structure. We do not want to dwell too much on the whereabouts of parametricity in this paper for the lack of space, so that we will not insist on that property and let the reader refer to the actual implementation (`https://github.com/CoqHott/coq-forcing/theories/yoneda.v`). □

Although this is not totally satisfying because of mismatches between type theory and category theory, note that in the special case where the base category is proof-irrelevant (i.e. a preorder) the translation actually builds an equivalent category.

Disregarding these small defects, we will consider that by applying the Yoneda translation to any category, we recover a new category which is essentially the same as the first one except that it has definitional equalities. By plugging it into the forcing translation, we will consequently fulfill all the expected conditions for the soundness theorems to go through.

5. Datatypes

We now proceed to extend the calculus with positives, that is datatypes defined by their constructors and move towards a translation of CIC. In CIC, datatypes are defined using a generic schema for declaring inductive types, using a generic eliminator construct for pattern-matching.

We wish to apply the forcing translation to any inductive definition, however there are a number of issues to resolve before doing so, having to do with dependent elimination. For the sake of conciseness, we will focus on Σ-types, whose definition is given in Figure 3. It is noteworthy to remark that we present Σ-types in a positive fashion, that is through pattern-matching, rather than negatively through projections. The latter is usually easier to interpret in an effectful setting, but it is weaker and in general does not extend to other types that have to be interpreted positively such as sums.

Whereas in the plugin our translation of inductive types builds new inductive types, for the sake of simplicity, we will directly translate Σ-types as Σ-types in this paper. There is little room left for tinkering. As the translation is by-name, we need to treat the subterms of pairs as application arguments by thunking them using the $[-]^!_\sigma$ macro and similarly for types.

Definition 17 (Forcing translation of Σ-types)**.**

$$\begin{array}{lcl} [\Sigma x : A.\, P]_\sigma & := & \lambda(q\ f : \sigma).\, \Sigma x : [\![A]\!]^!_{\sigma\cdot(q,f)}.\, [\![P]\!]^!_{\sigma\cdot(q,f)\cdot x} \\ [(M, N)]_\sigma & := & ([M]^!_\sigma, [N]^!_\sigma) \\ [\mathtt{match}\ M\ \mathtt{with}\ (x, y) \Rightarrow N]_\sigma & := & \\ \multicolumn{3}{r}{\mathtt{match}\ [M]_\sigma\ \mathtt{with}\ (x, y) \Rightarrow [N]_{\sigma\cdot x\cdot y}} \end{array}$$

Proposition 13. *The translation enjoys computational soundness.*

Against all expectations, typing soundness is not provable for the whole CIC. While the typing rules of formation, introduction and non-dependent elimination are still valid, the dependent elimination rule needs to be restricted. Indeed, the conclusion of the traditional dependent elimination rule for Σ-types is

$$\mathtt{match}\ M\ \mathtt{with}\ (x, y) \Rightarrow N : C\{z := M\}$$

This rule is not valid in presence of effects, because on the left-hand side, M is directly evaluated, whereas on the right-hand side, the evaluation of M is postponed. In particular, it is not valid in the forcing layer, and thus cannot be interpreted by the forcing translation. The translation of this sequent results effectively in

$$\mathtt{match}\ [M]_\sigma\ \mathtt{with}\ (x, y) \Rightarrow [N]_{\sigma\cdot x\cdot y} : [\![C]\!]_{\sigma\cdot z}\{z := [M]^!_\sigma\}$$

and it is clear that $[M]^!_\sigma$ can have little to do with $[M]_\sigma$. Intuitively, a *boxed* term—i.e. a term expecting a forcing condition before returning a value—of the translated inductive type can use the forcing conditions to build different inductive values at different conditions. It is for instance easy to build boxed booleans, i.e. terms of type $[\![\mathbb{B}]\!]^!_\sigma := \Pi(q\ f : \sigma).\, \mathbb{B}$ that are neither $[\mathtt{true}]^!_\sigma$ nor $[\mathtt{false}]^!_\sigma$ but whose value depends on the forcing conditions. There is hence no reason for it to be propositionally equal to a constructor application, let alone definitionally.

Therefore, we restrict the source type theory to dependent eliminations where a `match` has type `match`, forcing evaluation in the result type as well. We denote this restricted theory CIC^- and summarize its typing rules at Figure 3.

Proposition 14. *Typing soundness holds for the* CIC^- *rules.*

In this effectful setting, the usual dependent elimination of CIC can be decomposed into a restricted elimination followed by an η-rule for Σ-types which can be written:

$$\mathtt{match}\ M\ \mathtt{with}\ (x, y) \Rightarrow C\{z := (x, y)\} \equiv_\eta C\{z := M\}.$$

While this η-rule is actually propositionally valid in CIC, it is not preserved by the forcing translation and can be disproved using non-standard boxed terms. In general, assuming definitional η-rules for positive datatypes makes conversion checking hard, in particular for sum types, requiring commutative conversions and very elaborate algorithms even in the simply-typed case [13]. Of course CIC^- plus definitional η-rules for inductive datatypes is equivalent to CIC plus those same rules, but an exact correspondence between CIC^- and CIC is harder to pin down.

Note that the translation also applies directly to the hidden return type annotation found in CIC, which we did not expose here for simplicity. The same technique can be applied to any algebraic datatype.

$$A, B, M, N ::= \ldots \mid \Sigma x : A.\, B \mid (M, N) \mid \texttt{match}\ M\ \texttt{with}\ (x, y) \Rightarrow N$$

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

$$\frac{\Gamma \vdash M : \Sigma x : A.\, B \qquad \Gamma \vdash C : \square \qquad \Gamma, x : A, y : B \vdash N : C}{\Gamma \vdash \texttt{match}\ M\ \texttt{with}\ (x, y) \Rightarrow N : C}$$

$$\frac{\Gamma \vdash M : \Sigma x : A.\, B \qquad \Gamma, z : \Sigma x : A.\, B \vdash C : \square \qquad \Gamma, x : A, y : B \vdash N : C\{z := (x, y)\}}{\Gamma \vdash \texttt{match}\ M\ \texttt{with}\ (x, y) \Rightarrow N : \texttt{match}\ M\ \texttt{with}\ (x, y) \Rightarrow C\{z := (x, y)\}}$$

Figure 3. Typing rules for Σ-types in CIC^-

6. Recursive Types

The datatypes described in the previous section are all non-recursive. Handling general inductive datatypes raises issues of its own, because we need to be clever enough in the definition to preserve both syntactical typing and reduction rules.

We will define our translation into CIC without giving all the technical details usually imposed by recursive types, amongst others positivity condition and guardedness. The reader can assume a theory close to the one implemented by Coq and Agda for instance. Our practical implementation uses Coq, so that we will use its particular syntax.

Rather than giving the generic translation, which would turn out to be rather uninformative to the reader and too technical, we will focus instead on a running example.[2] This example should be rich enough to uncover the issues stemming from recursive types. We should stick to the `list` type, for it features a parameter. We recall that it is defined as follows.

$$\begin{array}{l}\texttt{Inductive list}\,(A : \square) : \square :=\\ \mid \texttt{nil} : \texttt{list}\ A\\ \mid \texttt{cons} : A \to \texttt{list}\ A \to \texttt{list}\ A\end{array}$$

The above definition generates the typing rules below, plus fixpoint and pattern-matching terms with the corresponding rules.

$$\frac{\Gamma \vdash A : \square}{\Gamma \vdash \texttt{list}\ A : \square} \qquad \frac{\Gamma \vdash A : \square}{\Gamma \vdash \texttt{nil}\ A : \texttt{list}\ A}$$

$$\frac{\Gamma \vdash A : \square \qquad \Gamma \vdash M : A \qquad \Gamma \vdash N : \texttt{list}\ A}{\Gamma \vdash \texttt{cons}\ A\ M\ N : \texttt{list}\ A}$$

6.1 Type and Constructor Translation

On the type itself, the translation is not that difficult. The only really subtle part arises from the forcing translation of types as we have

$$[\![\square_i]\!]_\sigma := \Pi(q\ f : \sigma).\, \square_i$$

so that the translation of an inductive type must take a forcing condition and a morphism as arguments.

Now, recursive types appear as arguments of their constructors, and following the by-name discipline, it means that they must be boxed. In particular, it implies that the forcing conditions change at each recursive invocation. There are a lot of possible design choices here when only following typing hints, but only one seems to comply with the reduction rules. It consists in enforcing the fact that the inductive does not depend on the current forcing conditions by simply not taking them as arguments and only rely on one condition.

[2] The Coq plugin translates any (mutually) inductive type.

Formally, we define an intermediate inductive $\texttt{list}^\bullet$, and the forcing translation for the `list` type is derived from it by applying it to the last forcing condition. The whole translation is defined in Figure 4. We use macros to show that the translation is straightforward, but they should really be thought of as their unfolding.

Proposition 15 (Typing soundness)**.** *The forcing translation of Figure 4 preserves the three typing rules of* `list`*,* `nil` *and* `cons`*.*

One important remark is that even though A is a uniform parameter of the `list` type, it is not anymore in its translation, because it is lifted to a future condition at each recursive call. Indeed, the type $[\![\texttt{list}\ A]\!]^!_{p \cdot A}$ in the recursive call in $\texttt{cons}^\bullet$ is convertible to

$$\Pi(q\ f : p).\, \texttt{list}^\bullet\ q\ (\lambda(r\ g : p \cdot (q, f)).\, A\ r\ (f \circ g))$$

where $\texttt{list}^\bullet$ has a different argument than A. This is not really elegant, but it does not cause more trouble than mere technicalities.

6.2 Non-dependent Induction

As in the non-recursive case, it is easy to define a non-dependent recursor on the translation of a recursive inductive type, because pattern-matchings are actually translated as pattern-matchings and similarly for fixpoints. For our running example, we can indeed build a function that folds over a forced list.

Definition 18 (Recursor)**.** A recursor for lists is a term `rec` of type

$$T_{\texttt{rec}} := \Pi(A\ P : \square).\, P_0 \to P_s \to \texttt{list}\ A \to P$$

with $P_0 := P$ and $P_s := A \to \texttt{list}\ A \to P \to P$ which is subject to the conversions

$$\begin{array}{ll}\texttt{rec}\ A\ P\ H_0\ H_s\ (\texttt{cons}\ A\ M\ N) & \equiv H_s\ M\ N\ (\texttt{rec}\ A\ P\ H_0\ H_s\ N)\\ \texttt{rec}\ A\ P\ H_0\ H_s\ (\texttt{nil}\ A) & \equiv H_0\end{array}$$

assuming the proper typing requirements.

Proposition 16 (Recursor Translation)**.** *Assuming a recursor* `rec`*, there exists a term* $\texttt{rec}^\bullet$ *of type* $\Pi p : \mathbb{P}.\, [\![T_{\texttt{rec}}]\!]_p$ *such that by posing*

$$[\texttt{rec}\ A\ P\ H_0\ H_s\ M]_\sigma := \texttt{rec}^\bullet\ \sigma_e\ [A]^!_\sigma\ [P]^!_\sigma\ [H_0]^!_\sigma\ [H_s]^!_\sigma\ [M]^!_\sigma$$

the forcing translation interprets the reduction rules of Definition 18 definitionally.

Proof. This recursor is built out of the actual recursor on $\texttt{list}^\bullet$ in a straightforward way. □

6.3 Storage Operators

Just as for the plain datatypes, dependent elimination is troublesome, because non-canonical terms can get in the way. It means that we cannot reasonably aim for the usual induction principles of inductive types, as we can simply disprove them by handcrafted

$$\begin{array}{l}\texttt{Inductive list}^\bullet\,(p:\mathbb{P})\,(A:[\![\Box]\!]^!_p)\,:\Box:=\\ \mid \texttt{nil}^\bullet:\texttt{list}^\bullet\,p\,A\\ \mid \texttt{cons}^\bullet:[\![A]\!]^!_{p\cdot A}\to[\![\texttt{list}\,A]\!]^!_{p\cdot A}\to\texttt{list}^\bullet\,p\,A\end{array}$$

$$\begin{array}{lcl}[\texttt{list}\,A]_\sigma & := & \lambda(q\,f:\sigma).\,\texttt{list}^\bullet\,q\,[A]^!_{\sigma\cdot(q,f)}\\ [\texttt{nil}\,A]_\sigma & := & \texttt{nil}^\bullet\,\sigma_e\,[A]^!_\sigma\\ [\texttt{cons}\,A\,M\,N]_\sigma & := & \texttt{cons}^\bullet\,\sigma_e\,[A]^!_\sigma\,[M]^!_\sigma\,[N]^!_\sigma\end{array}$$

Figure 4. List translation

terms. The situation is actually even direr, because trying to take a simple `match`-expansion trick is not enough to make the inductive case go through. We need something stronger.

Luckily, we came up with a restriction inspired from another context where forcing interacts with effects: classical realizability. In order to recover the induction principle on natural numbers in presence of `callcc`, Krivine introduced the notion of *storage operators* [8]. Essentially, a storage operator, e.g. for integers, is a term $\vartheta_{\mathbb{N}}$ of type $\mathbb{N}\to(\mathbb{N}\to R)\to R$ which purifies an integer argument by recursively evaluating and reconstructing it. The suspicious $(\mathbb{N}\to R)\to R$ return type is actually a trick to encode call-by-value in a call-by-name setting thanks to a CPS, so that we are sure that the integer passed to the continuation is actually a value.

Storage operators are somehow arcane outside of the realm of classical realizability, but they are actually both really simple to define from a recursor, computationally straightforward and invaluable to our translation. Once again, we only define here a storage operator for the `list` type but this can be generalized.

Definition 19 (Storage operator)**.** Assuming a recursor `rec`, we define the storage operator for lists ϑ in Figure 5. We will omit the A and R arguments when applying ϑ for brevity.

Storage operators are only defined by means of the non-dependent recursor, so they have a direct forcing translation by applying Proposition 16. Moreover, in a pure setting, they are pretty much useless, as the following proposition holds.

Proposition 17 (Propositional η-rule)**.** CIC *proves that*

$$\Pi(A\,R:\Box)\,(l:\texttt{list}\,A)\,(k:\texttt{list}\,A\to R).\,\vartheta\,l\,k=k\,l.$$

This is proved by a direct *dependent* induction over the list. This is precisely where the forcing translation fails, and the above theorem does not survive the forcing translation.

6.4 Dependent Induction in an Effectful World

By using storage operators, we can nevertheless provide the effectful equivalent of an induction principle on recursive types.

Theorem 3. *There exists a term* $\texttt{ind}^\bullet$ *of type* $\Pi p:\mathbb{P}.\,[\![T_{\texttt{ind}}]\!]_p$ *where*

$$\begin{array}{lcl}T_{\texttt{ind}} & := & \Pi(A:\Box)\,(P:\texttt{list}\,A\to\Box).\\ & & \quad P_0\to P_s\to\Pi l:\texttt{list}\,A.\,\vartheta\,l\,P\\ P_0 & := & P\,(\texttt{nil}\,A)\\ P_s & := & \Pi(x:A)\,(l:\texttt{list}\,A).\,\vartheta\,l\,P\to\vartheta\,(\texttt{cons}\,A\,x\,l)\,P\end{array}$$

which is subject to the conversion rules of Definition 18 (by replacing `rec` *by* `ind`*).*

Proof. Once again, it is a straightforward application of the dependent induction principle for $\texttt{list}^\bullet$. □

In the usual CIC, the above theorem seems to be a very contrived way to state the dependent induction principle. By rewriting the propositional η-rule, even its type is *equal* to the type of the usual induction principle. Yet, in the effectful theory resulting from the forcing translation, the two theorems are sharply distinct, as the usual induction principle is disprovable in general.

6.5 Revisiting the Non-Recursive Case

Actually, even the restriction on dependent elimination from Section 5 can be presented in terms of storage operators. As soon as a non-recursive type is defined by constructors, one can easily define storage operators over it by pattern-matching alone.

Definition 20 (Simple storage operator)**.** We define a storage operator ϑ_Σ for Σ-types in Figure 6.

It is now obvious that the `match` restriction when typing dependent pattern-matching corresponds exactly to the insertion of a storage operator in front of the type of the expression, i.e. the typing rule of Figure 6 is equivalent to the one of Section 5 up to conversion.

Therefore, we advocate for the use of storage operators as a generic way to control effects in a dependent setting. Purity is recovered by adding the η-law on storage operators as a theorem in the theory, or even definitionally. To the best of our knowledge, this use of storage operators is novel in a dependent type theory equipped with effects, notwithstanding the actual existence of such an object.

7. Forcing at Work: Consistency Results

In this section, we present preservation of (a simple version of) functional extensionality and the fact that the negation of the univalence axiom is compatible with CIC. Then, we show that (a simple version of) the univalence axiom is preserved for types which respect a monotonicity condition.

7.1 Equality in CIC

Before stating consistency result, we need to look at the notion of equality in CIC and in the forcing layer. As usual, equality in CIC is given by the inductive `eq` with one constructor `refl` as follows:

$$\begin{array}{l}\texttt{Inductive eq}\,(A:\Box)\,(x:A)\,:A\to\Box:=\\ \mid \texttt{refl}:\texttt{eq}\,A\,x\,x\end{array}$$

We simply write $x=y$ for $\texttt{eq}\,A\,x\,y$ when A is clear from context. Following the automatic translation of inductive types explained in Section 6, `eq` is translated as

$$\begin{array}{l}\texttt{Inductive eq}^\bullet\,(p:\mathbb{P})\,(A:[\![\Box]\!]^!_p)\,(x:[\![A]\!]^!_p)\,:[\![A]\!]^!_p\to\Box:=\\ \mid \texttt{refl}^\bullet:\texttt{eq}^\bullet\,p\,A\,x\,x\end{array}$$

Actually, we can show that the canonical function from $x=y$ to $\texttt{eq}^\bullet\,p\,A\,x\,y$ (obtained by eliminating over $x=y$) is an equivalence[3] for all forcing condition p. This means that the property satisfied by `eq` in the core calculus can be used to infer properties on `eq` in the forcing layer.

Using a storage operator, we can define a dependent elimination that corresponds to the J eliminator of Martin-Löf's type theory. Nevertheless, we simply need here the following Leibniz principle, which avoids the use of storage operators because the returned

[3] In homotopy type theory, being an equivalence is defined as the predicate

$$\begin{array}{l}\texttt{IsEquiv}:=\lambda(A\,B:\Box)\,(f:A\to B).\\ \quad\Sigma g:B\to A.\,(\Pi x.\,g\,(f\,x)=x)\times(\Pi y.\,f\,(g\,y)=y).\end{array}$$

$$\begin{array}{rcl}\vartheta & : & \Pi(A\,R:\square).\,\texttt{list}\;A \to (\texttt{list}\;A \to R) \to R \\ & := & \lambda(A\,R:\square).\,\texttt{rec}\;A\;((\texttt{list}\;A \to R) \to R) \\ & & \quad(\lambda k : \texttt{list}\;A \to R.\,k\;(\texttt{nil}\;A)) \\ & & \quad(\lambda(x:A)\,(\_:\texttt{list}\;A)\,(r:(\texttt{list}\;A \to R) \to R)\,(k:\texttt{list}\;A \to R).\,r\;(\lambda l : \texttt{list}\;A.\,k\;(\texttt{cons}\;A\;x\;l)))\end{array}$$

Figure 5. Storage operator for lists

$$\begin{array}{rcl}\vartheta_\Sigma & : & \Pi(A:\square)\,(B:A\to\square)\,(R:\square).\,(\Sigma x:A.\,B) \to ((\Sigma x:A.\,B) \to R) \to R \\ & := & \lambda(A:\square)\,(B:A\to\square)\,(R:\square)\,(p:\Sigma x:A.\,B)\,(k:(\Sigma x:A.\,B)\to R).\,\texttt{match}\;p\;\texttt{with}\;(x,y) \Rightarrow k\;(x,y)\end{array}$$

$$\frac{\Gamma \vdash M : \Sigma x:A.\,B \qquad \Gamma, z:\Sigma x:A.\,B \vdash C:\square \qquad \Gamma, x:A, y:B \vdash N : C\{z := (x,y)\}}{\Gamma \vdash \texttt{match}\;M\;\texttt{with}\;(x,y) \Rightarrow N : \vartheta_\Sigma\;M\;(\lambda z:\Sigma x:A.\,B.\,C)}$$

Figure 6. Storage operator for Σ-types

predicate does not depend on the equality:

$$\Pi A\,(x\,y:A)\,(P:A\to\square)\,(e:x=y).\,P\;x \to P\;y.$$

7.2 Preservation of Functional Extensionality

The precise statement of functional extensionality in homotopy type theory is that the term `f_equal` of type:

$$\Pi A\,(B:A\to\square)\,(\varphi\,\psi:\Pi x.\,B\;x).\,\varphi=\psi \to \Pi x.\,\varphi\;x=\psi\;x$$

is an equivalence. This term is obtained from Leibniz's principle and expresses that when two functions are equal, they are equal pointwise.

Assuming functional extensionality in the core calculus, we can define a weaker variant of functional extensionality.

Proposition 18 (Preservation of functional extensionality)**.** *Assuming functional extensionality in the core calculus, one can define a term of type*

$$\Pi A\,(B:A\to\square)\,(\varphi\,\psi:\Pi x.\,B\;x).\,(\Pi x.\,\varphi\;x=\psi\;x) \to \varphi=\psi$$

in the forcing layer.

Proof. Once translated in the core calculus, using the equivalence between `eq` and $\texttt{eq}^\bullet$, it remains to give a term of type $\varphi=\psi$ for all forcing condition p and φ and ψ in $[\![\Pi x:A.\,B\;x]\!]_p$, assuming a term X of type $[\![\Pi x.\,\varphi\;x=\psi\;x]\!]^!_p$. Now, φ and ψ are functions that expect a forcing condition q, a morphism f : $\texttt{Hom}\;p\;q$ and an argument $[\![A]\!]^!_{p\cdot(q,f)}$. Using functional extensionality in the core calculus, this amounts to show $\varphi\;q\;f\;x=\psi\;q\;f\;x$. This can be deduced by using `f_equal` on $X\;p\;q\;x$ and applying it to q and `id`. □

The preservation of the complete axiom of functional extensionality would require some additional naturality conditions (similar to parametricity) in the translation (see Section 7.5 for a discussion on this point).

In the same way, we can prove the preservation of the *Uniqueness of Identity Proof* axiom which says that any proof of $x=x$ is by reflexivity.

7.3 Negation of the Univalence Axiom

In homotopy type theory, Voevodsky's univalence axiom is expressed by saying that the canonical map `path_to_equiv` of type

$$A=B \to \Sigma\varphi : A\to B.\,\texttt{IsEquiv}\;A\;B\;\varphi$$

is an equivalence. This term is defined using Leibniz's principle on the identity equivalence. This axiom sheds light on the connection between CIC and homotopy theory—more specifically higher topos theory. This axiom expresses that the only way to observe a type is through its interaction with the environment. Actually, this axiom can be wrong in presence of effects because types may perform effects that cannot be observed because a type A is always observed uniformly at every possible future condition and not at a given one.

Proposition 19 (Negation of the univalence axiom)**.** *There exists a forcing layer in which the type*

$$(\Pi(A\,B:\square).\,\texttt{IsEquiv}\;\_\;\_\;(\texttt{path\_to\_equiv}\;A\;B)) \to \bot$$

can be inhabited.[4]

Proof. We define the forcing condition to be $\mathbb{P} := \texttt{bool}$ and for all p,q : `bool`, $\texttt{Hom}(p,q) := \texttt{unit}$ where `bool` (resp. `unit`) is the inductive type with two (resp. one) elements. In this layer, it is possible to define two new types (at level p)

$$\begin{array}{rcl}A_0 & := & \lambda(q\,f:p).\,\texttt{if}\;q\;\texttt{then}\;\texttt{unit}\;\texttt{else}\;\bot \quad : \quad [\![\square]\!]_p \\ A_1 & := & \lambda(q\,f:p).\,\texttt{if}\;q\;\texttt{then}\;\bot\;\texttt{else}\;\texttt{unit} \quad : \quad [\![\square]\!]_p\end{array}$$

Those two types are obviously different in the forcing layer. However, it is possible to define a function from A_0 to A_1 by using the fact that functions expect their arguments to be given for every possible future forcing condition. Thus, to define the function at condition, say p, one just has to use the argument at condition $\neg p$, the negation of p. Symmetrically, it is possible to define a function from A_1 to A_0, and to show that they form an equivalence. □

Note that the univalence axiom has been shown to be consistent with Martin-Löf's type theory using a simplicial model [7], which suggest the independence of the univalence axiom with CIC.

7.4 Preserving Univalence Axiom for Monotonous Types

In the previous section, we have been able to negate the univalence axiom by using types that produce completely non-monotonous effects. But if we restrict the univalence statement to types that respect a monotonicity condition, it becomes possible to prove the preservation of (a simple version of) univalence. Indeed, it is possible to define a modality $\bigcirc$ on $\square$ by

$$\bigcirc_p : [\![\square\to\square]\!]_p := \lambda X\,q\,f.\,\Pi r\,(g:\texttt{Hom}\;q\;r).\,X\;r\;(g\circ f)\;r\;\texttt{id}$$

We get a modality in the sense of [15][5]. A type A is $\bigcirc$-modal when it is equivalent to $\bigcirc A$. Those types are the types which satisfies

[4] $\bot$ is the inductive type with no constructor

[5] Up to a missing equality that can be recovered using naturality conditions of Section 7.5

a monotonicity condition. Restricting the univalence axiom to $\bigcirc$-modal types, we can recover (a simple form of) preservation of univalence.

Proposition 20 (Preservation of the univalence axiom for $\bigcirc$-modal types)**.** *Assuming univalence in the core calculus, one can define a term of type*

$$(\Sigma\varphi : A \to B.\, \mathtt{IsEquiv}\ A\ B\ \varphi) \to \bigcirc A = \bigcirc B$$

in the forcing layer.

Proof. The proof is similar to the proof of preservation of functional extensionality. It also uses the fact that assuming univalence in the core calculus also implies functional extensionality in the core calculus. The crux of the proof lies in the fact that A and B have only to be equal globally, and not pointwisely at each forcing condition.

For instance, the types A_0 and A_1 of Proposition 19 satisfy $(\bigcirc A_0) = (\bigcirc A_1)$. □

7.5 Towards Forcing with Naturality Conditions

Our forcing translation is much coarser than it could be, for it allows really non-standard terms that can abuse the forcing conditions a lot. Most notably, all boxed terms coming from the translation respect strong constraints that the current translation does not account for, and which are the call-by-name equivalent to the *naturality* requirement from the presheaf construction. For instance, all closed boxed types $A^\bullet : [\![\Box]\!]^!_\sigma \equiv \Pi(q\, f : \sigma)\,(r\, g : \sigma \cdot (q, f)).\,\Box$ verify the equality

$$A^\bullet\ q\ f\ r\ g \equiv A^\bullet\ r\ (f \circ g)\ r\ \mathtt{id}_r$$

for all q, f, r and g. The same goes for inductive types, as the need to restrict dependent elimination in CIC^- stems from the existence of boxed terms that allow themselves to observe the current conditions too much. By enforcing the fact that they must coincide at each later condition, we could recover a propositional η-rule and thus full dependent elimination.

Actually, it seems not that difficult to enforce such naturality properties by means of an additional bit of parametricity in the translation itself, in the style of Lasson [2]. Just as the call-by-value translation requires natural propositional equalities on the value types, we can do the same for values appearing in the CBPV decomposition of call-by-name, i.e. in the $[-]^!_\sigma$ and $[\![-]\!]^!_\sigma$ translations. This also means that the translation of each type A must embed a parametricity property $\Vdash_{A,\sigma} : [\![A]\!]^!_\sigma \to \Box$ specifying what it is to be natural at this type (i.e. parametric).

We believe that contrarily to the call-by-value forcing, this should not prevent the translation to preserve definitional equality. Indeed, as in the parametricity translation of PTS, we never rely on the additional equalities to compute, and merely pass them along the translation. Even more, the unary parametricity translation should probably be equivalent to the forcing translation with trivial conditions.

Such a translation would be in some sense purer. It would preserve the monotonous univalence axiom from the previous section, but also allow to prove propositionally the η-law for storage operators. Therefore, it would be the best of by-value and by-name forcing translations.

8. Conclusion and Future Work

In this paper, we have defined call-by-name forcing for the Calculus of Inductive Construction. It provides the first effectful translation of CIC into itself which preserves definitional equality and thus avoids the so-called coherence issue. The definition of inductives makes use of Krivine's storage operators to give rise to the first presentation of CIC with effects.

Our work allows to use any category to increase the logical power of CIC just as considering presheaves allows to increase the logical power of a topos. This is a first step towards the use of the category of cubes as the type of forcing conditions to give a computational content to the cubical type theory [4] of Coquand et al and in particular to the univalence axiom.

It also shed some new light on the difficult problem of combining dependent types with effects. Indeed, our translation is really close to a reader monad, the forcing conditions corresponding to some states that can be read, and locally modified in a monotonic way. It would be interesting to see if some of the techniques introduced here, notably the use of storage operators, could be applied to handle more general effects.

9. Acknowledgments

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

References

[1] T. Altenkirch and A. Kaposi. Type theory in type theory using quotient inductive types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016.

[2] J.-P. Bernardy and M. Lasson. Realizability and Parametricity in Pure Type Systems. In *Foundations of Software Science and Computational Structures*, volume 6604, pages 108–122, Saarbrücken, Germany, Mar. 2011. doi: 10.1007/978-3-642-19805-2.

[3] A. Brunel. *Transformations de «forcing» et algèbres de «monitoring»*. PhD thesis, 2014.

[4] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg. Cubical Type Theory: a constructive interpretation of the univalence axiom, 2015. Preprint.

[5] H. Herbelin. A constructive proof of dependent choice, compatible with classical logic. In *LICS*, pages 365–374. IEEE Computer Society, 2012.

[6] G. Jaber, N. Tabareau, and M. Sozeau. Extending Type Theory with Forcing. In *LICS 2012 : Logic In Computer Science*, pages 0–0, Dubrovnik, Croatia, June 2012.

[7] C. Kapulkin, P. L. Lumsdaine, and V. Voevodsky. The simplicial model of univalent foundations. *arXiv preprint arXiv:1211.2851*, 2012.

[8] J.-L. Krivine. Classical logic, storage operators and second-order lambda-calculus. *Ann. Pure Appl. Logic*, 68(1):53–78, 1994. doi: 10.1016/0168-0072(94)90047-7.

[9] J.-L. Krivine. Realizability in classical logic. *Panoramas et synthèses*, 27:197–229, 2009.

[10] P. B. Levy. *Call-by-push-value*. PhD thesis, Queen Mary, University of London, 2001.

[11] Z. Luo. ECC, an extended calculus of constructions. In *LICS*, pages 386–395. IEEE Computer Society, 1989.

[12] A. Miquel. Forcing as a program transformation. In *LICS*, pages 197–206. IEEE Computer Society, 2011. ISBN 978-0-7695-4412-0.

[13] G. Scherer. Multi-focusing on extensional rewriting with sums. In T. Altenkirch, editor, *13th International Conference on Typed Lambda Calculi and Applications, TLCA 2015, July 1-3, 2015, Warsaw, Poland*, volume 38 of *LIPIcs*, pages 317–331. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. ISBN 978-3-939897-87-3.

[14] M. Tierney. Sheaf theory and the continuum hypothesis. In *Toposes, algebraic geometry and logic*, pages 13–42. Springer, 1972.

[15] Univalent Foundations Project. *Homotopy Type Theory: Univalent Foundations for Mathematics*. `http://homotopytypetheory.org/book`, 2013.

[16] B. Werner. Sets in types, types in sets. In *Theoretical aspects of computer software*, pages 530–546. Springer, 1997.