The next 700 syntactical models of type theory

Simon Boulier, Pierre-Marie Pédrot, Nicolas Tabareau

The Next 700 Syntactical Models of Type Theory

Simon Boulier Pierre-Marie Pédrot Nicolas Tabareau

Inria, France

firstname.name@inria.fr

Abstract

A family of syntactic models for the calculus of construction with universes (CC_ω) is described, all of them preserving conversion of the calculus definitionally, and thus giving rise directly to a program transformation of CC_ω into itself. Those models are based on the remark that negative type constructors (e.g., dependent product, coinductive types or universes) are underspecified in type theory—which leaves some freedom on extra intensional specifications. The model construction can be seen as a compilation phase from a complex type theory into a simpler type theory. Such models can be used to derive (the negative part of) independence results with respect to CC_ω, such as functional extensionality, propositional extensionality, univalence or the fact that bisimulation on a coinductive type may not coincide with equality. They can also be used to add new principles to the theory, which we illustrate by defining a version of CC_ω with ad-hoc polymorphism that shows in particular that parametricity is not an implicit requirement of type theory. The correctness of some of the models/program transformations have been checked in the COQ proof assistant and have been instrumented as a COQ plugin.

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

Keywords Dependent type theory, Program translation

1. Introduction

"Can we prove that $\forall x.\, f\, x = g\, x \;\to\; f \;=\; g$?", "What is the difference between $P \leftrightarrow Q$ and $P = Q$, when P and Q are two propositions?", "Can we prove that a term of type $\forall A.\, A \to A$ is necessarily the identity function?". Those simple questions are frequently asked to someone who starts teaching the use of a theorem prover based on type theory such as COQ (The COQ Development Team 2015) or AGDA (The AGDA Development Team 2015). However, the definitive answer to those questions is far from being simple as it involves proving the consistency of a type theory augmented with some logical principles or their negation.

Traditionally, given a type theory described as a syntactic object, there are two ways to justify it.

1. The most standard one, coming all the way back from model theory, consists in building a model of it, usually into set theory or a given categorical structure. This amounts formally to translating the syntax into some well-behaved quotient. See e.g. (Werner 1997).
2. The most syntactical one consists in showing some good properties of the syntax, usually strong normalization and subject reduction, so as to derive the desired properties on the type theory.

Each approach suffers from its own set of defects. Models of dependent type theory have been widely studied from a set theoretic point of view (see (Hoffman 1997) for a general overview), giving rise to several families of models, e.g., categories with families (Dybjer 1996), comprehension categories (Jacobs 1993), ... But all those models of a type theory are built into some other fundational theory, putting back the burden on the consistency on the latter. This is unavoidable, because of the incompleteness theorem, and may be actually seen as an advantage when targeting such a stress-tested theory as set theory. Nonetheless, the discrepancy between the source theory and the target theory can be at the root of subtle but very important mismatches. This is typically seen when considering the decidability of conversion which may hold in the source theory, but is lost as soon as objects are interpreted semantically.

Going the syntactical way is not without issues either, as the proofs are often very tedious, not modular, and keep repeating themselves between close flavours of type theory.

In this paper, we advocate for a third alternative approach, based on a proof-as-program interpretation of type theory, which can somehow be seen as a reconciliation of the two aforementioned kinds of justification.

"To justify a complex type theory, it suffices to compile it into a simpler—already justified—type theory."

Modified Type	Name of the translation	New principles	Translation
$\Pi x : A.\, B$	Intensional functions (ext)	$\neg$ functional extensionality	$[\![\Pi x : A.\, B]\!] := (\Pi x : [\![A]\!].\, [\![B]\!]) \times \mathbb{B}$
	Intensional functions (int)	$\neg$ functional extensionality	$[\![\Pi x : A.\, B]\!] := \Pi x : [\![A]\!].\, [\![B]\!] \times \mathbb{B}$
$\mathtt{stream}\, A$	Intensional streams	$\neg$ stream extensionality	$[\![\mathtt{stream}\, A]\!] := \mathtt{stream}\, [\![A]\!] \times \mathbb{B}$
$*$	Ad-hoc intensional propositions	$\neg$ propositional extensionality	$[\![*]\!] := * \times \mathbb{B}$
$\square_i$	Ad-hoc intensional types	$\neg$ univalence	$[\![\square_i]\!] := \square_i \times \mathbb{B}$
	Ad-hoc Polymorphism	(quote) à la Lisp	$[\![\square_i]\!] := \mathtt{TYPE}_i$ *(inductive-recursive type)*
	parametricity	Built-in parametricity	$[\![\square_i]\!] := \lambda(A_0\, A_1 : \square_i).\, A_0 \to A_1 \to \square_i$
	Functional Reactive Programming	General Fixpoints	$[\![\square_i]\!] := \mathtt{stream}\, \square_i$
	Forcing over $\mathcal{P}$	*(depending on $\mathcal{P}$)*	more complicated, see (Jaber et al. 2016)

Figure 1. Different Translations of CC_ω

Essentially, we follow the model approach, but instead of targeting an alien system, we use type theory itself and can thus rely on the consistency of it, proved in a syntactical manner. Furthermore, contrarily to recent models of type theory into itself (Chapman 2009; Altenkirch and Kaposi 2016), we do not describe the source theory as a deep embedding in the target one, but rather use a shallow embedding where all source proofs are translated into the target system in a way that preserves most of the structure. This is no more than the dependently typed counterpart of more standard logical translations seen as program translations, e.g. Lafont-Streicher-Reus CPS (Lafont et al. 1993) is the programming language equivalent of some variant of Gödel's double-negation translation.

This way, several properties from the target theory are inherited for free, as long as the program transformation preserves conversion of the calculus definitionally and that the empty type $\perp$ (encoded here as $\Pi A : \square_i.\, A$) is translated into some type equivalent to $\perp$. This provides a very simple notion of model of type theory, accessible to a broader audience than specialists of category theory and set theory, which can be used to quickly answer if an axiom is derivable.

As said, such translations can not be used directly to prove the consistency of the type theory under consideration, and rely on the consistency of the base type theory itself. But they can rather be used to study some properties of type theory enriched with new principles. This approach is in accordance with the recent work on forcing in type theory (Jaber et al. 2012, 2016) and parametricity in type theory (Bernardy et al. 2010; Bernardy and Moulin 2012), which can both be seen as particular cases of the general picture described in this paper.

We provide several translations in this paper. They are based on the remark that negative type constructors (e.g., dependent product, coinductive types or universes) are underspecified in type theory—which leaves a lot of freedom on extra intensional specifications. Such translations can be used to derive a new type theory where (the negative part of) independence results can be proven, such as functional extensionality, propositional extensionality, univalence or the fact that bisimulation on a coinductive type may not coincide with equality (that will be called here *"stream extensionality"*).

It has to be noticed that obtaining similar results using set-theoretic models is very tedious. Indeed, although most available models negate univalence, finding a model that negates functional extensionality or propositional extensionality is much more difficult as those principles are hardcoded in set theory. It becomes even harder when it comes to models integrating coinductive types, as such model are based upon the notion of final coalgebra and analysing the connection between bisimilulation and equality in this setting is a very complex task.

Type-theoretic translations can also be used to add new principles to the theory, which we illustrate by defining a version of a type theory with ad-hoc polymorphism (i.e., enriched with the *(quote)* operator of Lisp) that shows in particular that parametricity is not an implicit requirement of type theory.

The systems we study are built upon CC_ω, a type theory featuring only dependent products and a denumarable hierarchy of universes $\square_i$.

Contributions. There are a handful of program translations in type theory (starting from the subset model of (Hofmann 1997)), but they are usually not presented as such. This is why we believe that giving a proper account of them is worthwhile, as it permits to exhibit the nice properties of those models, most notably the preservation of computation. We also hope to popularize such a way of building models.

In this paper, we provide the following contributions, which are summarized in Figure 1:

- we provide a general approach to defining translations of CC_ω into itself by adding more intensional properties to *negative* types

- we apply it to dependent products to show formally that functional extensionality is not derivable in intensional type theory
- we apply it to coinductive types to show that bisimilarity of streams does not imply in general equality of streams
- we apply it to the universe, by using a presentation *à la Tarski*, to show that univalence and even propositional extensionality are not derivable in CC_ω
- finally, we introduce a more complex interpretation of universes using induction-recursion to equip CC_ω with ad-hoc polymorphism and a quote operator

Coq Formalization. The correctness of the first three program translations have been checked in the COQ proof assistant and have been instrumented as a COQ plugin. They have been developed using the 8.5 release of COQ (The COQ Development Team 2015) and are available at:

`https://github.com/CoqHott/Program-translations-CC-omega`

2. Overview

The systems we study in this paper are built upon CC_ω, a type theory featuring only dependent products, also known as Π-types. It features a denumerable hierarchy of universes $\square_i$, and is a close relative to Luo's ECC (Luo 1989).

Definition 1 (Typing system)**.** We define two statements mutually recursively as usual. The statement $\vdash \Gamma$ means that the environment Γ is well formed, while $\Gamma \vdash M : A$ means that the term M has type A in environment Γ. The typing rules are given in Figure 2. $\square$ stands for any $\square_i$ or the impredicative universe $*$ when it is considered.

Before exposing specific translations between type theories, we sketch the general schema followed by those translations. Given two type theories, the source theory $\mathcal{S}$ and the target theory $\mathcal{T}$, we require that any term t in $\mathcal{S}$ is translated into a term $[t]$ in $\mathcal{T}$ by induction over the syntax of t, and that there is an internal operation ι which coerces a translated type into a type of the target type theory. We write $[\![A]\!]$ for $\iota\ [A]$. Note that the induced type may live in a bigger universe, we will use this fact in Section 5.2. The context is then translated pointwise as:

$$\begin{array}{lcl} [\![\cdot]\!] & := & \cdot \\ [\![\Gamma, x : A]\!] & := & [\![\Gamma]\!], x : [\![A]\!] \end{array}$$

The expected theorem is *typing soundness* which states that $[\![\Gamma]\!] \vdash [M] : [\![A]\!]$ whenever $\Gamma \vdash M : A$. This requires in particular to show a form of *computational soundness* which says that $[M] \equiv [N]$ whenever $M \equiv N$ in order to interpret the conversion rule transparently as another conversion. Another expected property of the translation is *consistency preservation*, saying that the translation of $\Pi X : \square_i . X$ is not inhabited. Typing soundness together with consistency preservation ensures consistency of the source theory provided consistency of the target theory.

We argue that this approach is strictly sharper than the various notions of models from the literature. First, the translation is only defined by induction on raw syntax, i.e. it does not depend on any typing assumption and is purely local. Furthermore, preservation of conversion gives a first-class treatment of computation even in an untyped setting. Finally, the target is pure type theory, so that every assumption needed to make the soundness proofs go through has to be internal, meaning that one cannot rely on pushing back side-conditions in the metatheory. We call this approach monistic, in the sense that it only requires type theory as foundations. Although a few instances of models of type theory from the literature can be described in this formalism, most of them cannot, and even those which can are typically not presented this way and rely on an intermediate structure such as categories with families or sets. Examples of models which are syntactical but not program translations include the setoid models of (Hofmann 1997) and (Altenkirch 1999). Amongst the few members of our class of models, one can cite the subset type (Paulin-Mohring 1989) and parametricity translations (Bernardy and Lasson 2011), which are quite similar, as well as the call-by-name forcing translation (Jaber et al. 2016).

The source theory $\mathcal{S}$ may contain more type constructors or logical properties than $\mathcal{T}$ as long as there is a term in $\mathcal{T}$ corresponding to the translation of those new type constructors or logical properties. In practice, $\mathcal{S}$ is often $\mathcal{T}$ + axiom and the translation ensures that the axiom can be assumed consistently. In this case, we also have a trivial translation from $\mathcal{T}$ to $\mathcal{T}$ + axiom (the injection) and hence the two theories are equiconsistent. Computational soundness can sometimes be refined so that we can deduce strong normalization for $\mathcal{S}$ from strong normalization for $\mathcal{T}$, although we will not describe this here. This is done by analyzing the preservation of reduction steps rather than mere conversion.

In all the translations presented in this paper, we use the fact that inhabitants of negative types can only be *partially* observed from the outside. For instance, dependent functions can only be observed using application. In the same way, types can only be observed on a right-hand side of a typing judgment, which means that the translation only depends on $[\![A]\!]$ and is oblivious to any additional data contained in $[A]$.

Figure 1 summarizes how this general scheme can be applied to various extensions of CC_ω to give a computational meaning to new principles available in those extensions. Those translations are presented in this paper, but for parametricity (already described in (Bernardy and Moulin 2012)) and for functional reactive programming and forcing which have already been described in (Jaber et al. 2012, 2016)—the former being a particular case of the latter because $\texttt{stream}\ A \simeq \mathbb{N} \to A$ corresponds to forcing on the

$$\begin{array}{rcl} \Gamma, \Delta & ::= & \cdot \mid \Gamma, x : A \\ A, B, M, N & ::= & \Box_i \mid x \mid M\ N \mid \lambda x : A.\, M \mid \Pi x : A.\, B \end{array}$$

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

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

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

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

Figure 2. Typing rules of CC_ω

pre-order of natural numbers. Those translations can be composed to add different principles together.

Extensions. Depending on the translation, we will use various extensions of the base CC_ω system, and we will refer to them explicitely when needed. Those extensions are summarized in Figure 3. They feature an impredicative sort $*$, as it exists in Coq, that will be used to distinguish between propositional extensionality and univalence. To reason about equality in CC_ω, we use Martin-Löf identity type, that is typed in $\Box_i$ (but it could be typed in $*$ as in Coq). We may need booleans and negative pairs, also known as Σ-types, with projections[1] and, in one case, surjective pairing.

Plan of the paper. In Section 3, we give a first translation that acts on dependent products in order to realize the negation of functional extensionality. This shows for instance that this principle is not provable in CC_ω. In Section 4, we show that a similar translation can be done on streams in order to negate the principle of stream extensionality. In Section 5, we provide two translations, one that negates univalence and even propositional extensionality and one that introduces a notion of ad-hoc polymorhism in the form of a (quote) operator à la Lisp. In Section 6, we show how some translations have been formalized in the Coq proof assistant and describe the Coq plugins that implement those translations.

3. Intensional Functions

In this section, we present a translation that augments every dependent function with a boolean value that cannot be seen externally as the only way to observe a function is by applying it, which forgets about the boolean value. In this translation, every function coming from a translated term of CC_ω will be equipped with the value `true`.

[1] We do not use a more general form of dependent elimination to simplify the theoretical development in this paper.

Definition 2. The intensional function translation $[\cdot]_f$ from CC_ω to $\mathrm{CC}_\omega + \Sigma + \mathbb{B}$ is defined by induction on terms as follows.

$$\begin{array}{lcl} [\Box_i]_f & := & \Box_i \\ [x]_f & := & x \\ [\lambda x : A.\, M]_f & := & (\lambda x : [\![A]\!]_f.\, [M]_f, \mathtt{true}) \\ [M\ N]_f & := & \pi_1\ [M]_f\ [N]_f \\ [\Pi x : A.\, B]_f & := & (\Pi x : [\![A]\!]_f.\, [\![B]\!]_f) \times \mathbb{B} \\ [\![A]\!]_f & := & [A]_f \end{array}$$

Note that $[\![A]\!]_f := [A]_f$, which means that for this translation, ι_f is the identity function, which is possible because $[\Box_i]_f := \Box_i$.

Proposition 3 (Substitution lemma)**.** *For all terms M, N and any variable x we have*

$$[M\{x := N\}]_f \equiv [M]_f\{x := [N]_f\}$$

Proof. By induction on M. ☐

Using the substitution lemma, it is not difficult to derive the two main properties of the translation.

Theorem 4 (Computational soundness)**.** *If $M \equiv N$ then $[M]_f \equiv [N]_f$.*

Proof. The only non-trivial case is the β-reduction rule, which is proved by the following rewriting steps.

$$\begin{array}{rcl} [(\lambda x : A.\, M)\ N]_f & := & \pi_1\ ((\lambda x : [\![A]\!]_f.\, [M]_f), \mathtt{true})\ [N]_f \\ & \equiv & (\lambda x : [\![A]\!]_f.\, [M]_f)\ [N]_f \\ & \equiv & [M]_f\{x := [N]_f\} \\ & \equiv & [M\{x := N\}]_f \end{array}$$

All congruence rules are interpreted as-is. ☐

Theorem 5 (Typing soundness)**.** *If $\Gamma \vdash M : A$ then $[\![\Gamma]\!]_f \vdash [M]_f : [\![A]\!]_f$.*

Impredicative Universe

$$A, B, M, N \quad ::= \quad \ldots \mid *$$

$$\frac{\vdash \Gamma}{\Gamma \vdash * : \square_i} \qquad \frac{\Gamma \vdash A : \square \qquad \Gamma, x : A \vdash B : *}{\Gamma \vdash \Pi x : A.\, B : *}$$

Negative Pairs

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

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

$$\frac{\Gamma \vdash M : \Sigma x : A.\, B}{\Gamma \vdash \pi_1\ M : A} \qquad \frac{\Gamma \vdash M : \Sigma x : A.\, B}{\Gamma \vdash \pi_2\ M : B\{x := \pi_1\ M\}}$$

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

Booleans

$$A, B, M, N \quad ::= \quad \ldots \mid \mathbb{B} \mid \texttt{true} \mid \texttt{false} \mid \texttt{if}\ M\ \texttt{ret}\ P\ \texttt{then}\ N_1\ \texttt{else}\ N_2$$

$$\frac{\Gamma \vdash M : \mathbb{B} \qquad \Gamma \vdash P : \mathbb{B} \to \square_i \qquad \Gamma \vdash N_1 : P\ \texttt{true} \qquad \Gamma \vdash N_2 : P\ \texttt{false}}{\Gamma \vdash \texttt{if}\ M\ \texttt{ret}\ P\ \texttt{then}\ N_1\ \texttt{else}\ N_2 : P\ M}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathbb{B} : \square_i} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \texttt{true} : \mathbb{B}} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \texttt{false} : \mathbb{B}}$$

$$\texttt{if}\ \texttt{true}\ \texttt{ret}\ P\ \texttt{then}\ N_1\ \texttt{else}\ N_2 \equiv N_1 \quad \texttt{if}\ \texttt{false}\ \texttt{ret}\ P\ \texttt{then}\ N_1\ \texttt{else}\ N_2 \equiv N_2$$

Equality

$$A, B, M, N \quad ::= \quad \ldots \mid \texttt{refl}_x \mid x =_A y \mid J(P, e, t)$$

$$\frac{\vdash \Gamma \qquad \Gamma \vdash A : \square_i \qquad \Gamma \vdash t, t' : A}{\Gamma \vdash t =_A t' : \square_i} \qquad \frac{\Gamma \vdash t : A}{\Gamma \vdash \texttt{refl}_t : t =_A t}$$

$$\frac{\Gamma \vdash e : t =_A t' \qquad \Gamma, y : A, q : t =_A t' \vdash P : \square_i \qquad \Gamma \vdash u : P\{y := t\}\{q := \texttt{refl}_t\}}{\Gamma \vdash J(P, e, u) : P\{y := t'\}\{q := e\}}$$

$$J(P, \texttt{refl}_t, u) \equiv u$$

Figure 3. Extensions to CC_ω

Proof. By induction on the typing derivation. The only non-trivial cases are the conversion rule which is interpreted by the above theorem, and rules manipulating products which are easily proved. □

This translation can account for various extensions of CC_ω in a straightforward fashion. For instance, we can interpret inductive datatypes functorially, even though we will not describe it here. In the end of this section, we assume we have at our disposal equality.

Theorem 6 (Consistency preservation)**.** *The translation preserves consistency.*

Proof. Indeed, we have

$$[\![\Pi A : \square_i.\, A]\!]_f := (\Pi A : \square_i.\, A) \times \mathbb{B}$$

which is inhabited if and only if $\Pi A : \square_i.\, A$ is. □

While being quite simple, this syntactical model already implements strictly more than just CC_ω. Thanks to the additional boolean worn by functions, it becomes possible to

$$
\begin{aligned}
&[\texttt{stream}\ A]_s := \texttt{stream}\ [\![A]\!]_s \times \mathbb{B}\\
&[\texttt{hd}\ M]_s := \texttt{hd}\ (\pi_1\ [M]_s)\\
&[\texttt{tl}\ M]_s := (\texttt{tl}\ (\pi_1\ [M]_s), \pi_2\ [M]_s)\\
&[\texttt{stream\_corec}\ S\ M_0\ M_1\ N]_s :=\\
&\qquad\qquad (\texttt{stream\_corec}\ [S]_s\ [M_0]_s\ [M_1]_s\ [N]_s, \texttt{true})\\
&[\texttt{bisim}\ A\ M\ N]_s := \texttt{bisim}\ [\![A]\!]_s\ (\pi_1\ [M]_s)\ (\pi_1\ [N]_s)\\
&[\texttt{hd}_{\texttt{b}}\ M]_s := \texttt{hd}_{\texttt{b}}\ [M]_s\\
&[\texttt{tl}_{\texttt{b}}\ M]_s := \texttt{tl}_{\texttt{b}}\ [M]_s\\
&[\texttt{bisim\_corec}\ S\ M_0\ M_1\ P_0\ P_1\ N]_s :=\\
&\qquad \texttt{bisim\_corec}\\
&\qquad\quad (\lambda s_0\, s_1.\, [S]_s\ (s_0, (\pi_2\ [P_0]_s))\ (s_1, (\pi_2\ [P_1]_s)))\\
&\qquad\quad (\lambda s_0\, s_1\, s.\, [M_0]_s\ (s_0, (\pi_2\ [P_0]_s))\ (s_1, (\pi_2\ [P_1]_s))\ s)\\
&\qquad\quad (\lambda s_0\, s_1\, s.\, [M_1]_s\ (s_0, (\pi_2\ [P_0]_s))\ (s_1, (\pi_2\ [P_1]_s))\ s)\\
&\qquad\quad (\pi_1\ [P_0]_s)\ (\pi_1\ [P_1]_s)\ [N]_s\\
&[\ldots]_s := \ldots\\
&[\![A]\!]_s := [A]_s
\end{aligned}
$$

Figure 4. Definition of the intensional stream translation

discriminate between two functions which are extensionally equal, thus negating functional extensionality. For the rest of this section we add identity types (Figure 3) in source and target theories. The translation extends directly by functoriality.

Theorem 7. *We define functional extensionality* `funext` *as follows.*

$$
\begin{aligned}
\texttt{funext} :=\ & \Pi(A : \square)\,(B : \square)\,(f\, g : A \to B).\\
& (\Pi x : A.\, f\ x =_B g\ x) \to f =_{A \to B} g
\end{aligned}
$$

Then there is, in the target system, a closed proof of $[\![\texttt{funext} \to \bot]\!]_f$.

Proof. It is sufficient to instantiate in the premise A and B by some arbitrary type X and f and g respectively by $(\lambda x : X.\, x, \texttt{true})$ and $(\lambda x : X.\, x, \texttt{false})$. This immediately leads to a proof that $\texttt{true} = \texttt{false}$ by injection and thus a contradiction. $\square$

Almost every model of CC_ω, if not all, validates `funext`, so it is consistent with CC_ω. The previous theorem shows that it is not provable in CC_ω.

Corollary 8. `funext` *is independent from* CC_ω.

It is possible to define a similar translation by adding the boolean value "inside" the function $[\Pi x : A.\, B]_f := \Pi x : [\![A]\!]_f.\, [\![B]\!]_f \times \mathbb{B}$. This translation is very similar so we do not present it in detail.

4. Intensional Coinductive Types

We now enrich our source theory with coinductive datatypes, and by using a translation similar to the one of intensional functions, we obtain in a straightforward way a variant for intensional streams. Using intensional streams, it is possible to prove that stream extensionality (the fact that bisimilarity coincides with equality for streams) is not provable in CC_ω.

Streams. We assume that our source and target theory are equipped with a type of streams and a relation of bisimilarity, defined in Figure 5. Given a stream M, it is possible to get its first element (or head) by using the destructor `hd` M and to get its tail by using the destructor `tl` M. The only way to create a stream is by coinduction[2] using the constructor `stream_corec` which produces, for any type S, a map from S to `stream` A given a map $S \to A$ and a map $S \to S$. Bisimilarity of streams is defined similarly.

The translation for intensional streams works by adding a boolean value to every stream, throwing it away when accessing the head, and threading it when accessing the tail.

Definition 9. The intensional stream translation $[\cdot]_s$ from $\mathrm{CC}_\omega + \texttt{stream}$ into $\mathrm{CC}_\omega + \Sigma + \mathbb{B} + \texttt{stream}$ is defined by induction on terms as defined in Figure 4, where unspecified cases are defined by commutation of the translation with the corresponding syntax constructor.

The properties satisfied by $[\cdot]_s$ are the same as for intensional functions, the proof arguments being similar.

Theorem 10 (Typing soundness)**.** *Assuming definitional surjective pairing in the target, if* $\Gamma \vdash M : A$ *then* $[\![\Gamma]\!]_s \vdash [M]_s : [\![A]\!]_s$.

Proof. Essentially the same as for the functional case. Conversion is once again interpreted as conversion, as the substitution lemma is trivial. We only need surjective pairing in order to show that $[N]_s$ is well-typed in the interpretation of `bisim_corec`, otherwise all rules are straightforward. $\square$

Remark 11. The requirement of surjective pairing can be lifted by defining a new dedicated coinductive type in the target system. We choose instead to reuse the same notion of bisimilarity for the sake of conceptual simplicity at the expense of a stronger requirement on the target system.

Even more directly than for dependent functions, falsity is translated into itself because the translation does not act on dependent products, so that we can derive consistency preservation directly.

Theorem 12 (Consistency preservation)**.** *The translation preserves consistency.*

As coinductive types are negatives types, the only way to observe them is by using their destructors, which for the case of streams amounts to observing only the head and the tail of a stream (coinductively). Thus, intensional streams allow to

[2] Coinductive types are defined categorically as terminal coalgebras, so their universal property can be used to construct maps going into them. This corresponds to the coinduction principle.

$$\begin{array}{rcl} A, B, M, N & ::= & \ldots \mid \texttt{stream}\ A \mid \texttt{hd}\ M \mid \texttt{tl}\ M \mid \texttt{stream\_corec}\ S\ M_0\ M_1\ N \\ & & \mid \texttt{bisim}\ A\ M\ N \mid \texttt{hd}_\texttt{b}\ M \mid \texttt{tl}_\texttt{b}\ M \mid \texttt{bisim\_corec}\ S\ M_0\ M_1\ P_0\ P_1\ N \end{array}$$

$$\frac{\Gamma \vdash S : \Box_i \qquad \Gamma \vdash M_0 : S \to A \qquad \Gamma \vdash M_1 : S \to S \qquad \Gamma \vdash N : S}{\Gamma \vdash \texttt{stream\_corec}\ S\ M_0\ M_1\ N : \texttt{stream}\ A}$$

$$\frac{\Gamma \vdash S : \texttt{stream}\ A \to \texttt{stream}\ A \to \Box_i \qquad \Gamma \vdash M_0 : T_0 \qquad \Gamma \vdash M_1 : T_1 \qquad \Gamma \vdash N : S\ P_0\ P_1}{\Gamma \vdash \texttt{bisim\_corec}\ S\ M_0\ M_1\ P_0\ P_1\ N : \texttt{bisim}\ A\ P_0\ P_1}$$

$$\text{where} \quad \begin{array}{rcl} T_0 & := & \Pi(s_0\, s_1 : \texttt{stream}\ A).\, S\ s_0\ s_1 \to \texttt{hd}\ s_0 =_A \texttt{hd}\ s_1 \\ T_1 & := & \Pi(s_0\, s_1 : \texttt{stream}\ A).\, S\ s_0\ s_1 \to S\ (\texttt{tl}\ s_0)\ (\texttt{tl}\ s_1) \end{array}$$

$$\frac{\Gamma \vdash A : \Box_i}{\Gamma \vdash \texttt{stream}\ A : \Box_i} \qquad \frac{\Gamma \vdash A : \Box_i \qquad \Gamma \vdash M : \texttt{stream}\ A \qquad \Gamma \vdash N : \texttt{stream}\ A}{\Gamma \vdash \texttt{bisim}\ A\ M\ N : \Box_i}$$

$$\frac{\Gamma \vdash M : \texttt{stream}\ A}{\Gamma \vdash \texttt{hd}\ M : A} \qquad \frac{\Gamma \vdash M : \texttt{stream}\ A}{\Gamma \vdash \texttt{tl}\ M : \texttt{stream}\ A} \qquad \frac{\Gamma \vdash M : \texttt{bisim}\ A\ N_1\ N_2}{\Gamma \vdash \texttt{hd}_\texttt{b}\ M : \texttt{hd}\ N_1 =_A \texttt{hd}\ N_2} \qquad \frac{\Gamma \vdash M : \texttt{bisim}\ A\ N_1\ N_2}{\Gamma \vdash \texttt{tl}_\texttt{b}\ M : \texttt{bisim}\ A\ (\texttt{tl}\ N_1)\ (\texttt{tl}\ N_2)}$$

$$\texttt{hd}\ (\texttt{stream\_corec}\ S\ M_0\ M_1\ N) \equiv M_0\ N \qquad \texttt{tl}\ (\texttt{stream\_corec}\ S\ M_0\ M_1\ N) \equiv \texttt{stream\_corec}\ S\ M_0\ M_1\ (M_1\ N)$$

$$\texttt{hd}_\texttt{b}\ (\texttt{bisim\_corec}\ S\ M_0\ M_1\ P_0\ P_1\ N) \equiv M_0\ P_0\ P_1\ N$$

$$\texttt{tl}_\texttt{b}\ (\texttt{bisim\_corec}\ S\ M_0\ M_1\ P_0\ P_1\ N) \equiv \texttt{bisim\_corec}\ S\ M_0\ M_1\ (\texttt{tl}\ P_0)\ (\texttt{tl}\ P_1)\ (M_1\ P_0\ P_1\ N)$$

Figure 5. Coinductive types

negate stream extensionality by comparing two streams that share the same elements, but differ on the additional boolean value. As previous, the translation extends to identity types and we have:

Theorem 13. *We define stream extensionality* `streamext` *as follows.*

$$\texttt{streamext} := \Pi(A : \Box)\ (s_1\ s_2 : \texttt{stream}\ A).\ \texttt{bisim}\ A\ s_1\ s_2 \to s_1 =_{\texttt{stream}\ A} s_2$$

Then there is in the target system a closed proof of $[\![\texttt{streamext} \to \bot]\!]_s$.

Proof. Once again, as in the functional case, intensional equality observes the boolean in the translated streams but bisimilarity does not. □

5. Intensional Types

In this section, we exploit the underspecification of types in CC_ω to give more content to types. Our first translation builds upon the $\cdot \times \mathbb{B}$ trick we used in the previous sections, but our second translation is much more subtle as it allows to implement ad-hoc polymorphism in vanilla type theory.

5.1 Ad-hoc intensional types

We define here a translation giving a very naive intensional content to types. Just as we did in the function case, we enrich every type with a boolean which is not observable in the source theory, except for intensional equality.

Definition 14. The ad-hoc intensional type translation $[\cdot]_t$ from CC_ω to $\mathrm{CC}_\omega + \Sigma + \mathbb{B}$ is defined by induction on terms as follows.

$$\begin{array}{lcl} [\Box_i]_t & := & (\Box_i \times \mathbb{B}, \texttt{true}) \\ [x]_t & := & x \\ [\lambda x : A.\, M]_t & := & \lambda x : [\![A]\!]_t.\, [M]_t \\ [M\ N]_t & := & [M]_t\ [N]_t \\ [\Pi x : A.\, B]_t & := & (\Pi x : [\![A]\!]_t.\, [\![B]\!]_t, \texttt{true}) \\ [\![A]\!]_t & := & \pi_1\ [A]_t \end{array}$$

As in previous sections, the translation verifies the substitution lemma and computational soundness, from which we derive:

Theorem 15 (Typing soundness)**.** *If* $\Gamma \vdash M : A$ *then* $[\![\Gamma]\!]_t \vdash [M]_t : [\![A]\!]_t$.

Proof. By induction on the typing derivation of $\Gamma \vdash M : A$. The only interesting cases are the two rules for type constructors.

1. Case $\Box_i : \Box_j$. Direct from the fact that

$$(\Box_i \times \mathbb{B}, \texttt{true}) : \Box_j \times \mathbb{B}$$

2. Case $\Pi x : A.\, B : \Box_j$. Direct from the fact that

$$(\Pi x : [\![A]\!]_t.\, [\![B]\!]_t, \texttt{true}) : \Box_i \times \mathbb{B}$$

□

Theorem 16 (Consistency preservation)**.** *The translation preserves consistency.*

Proof. This comes immediately from the fact that

$$[\![\Pi A : \Box_i .\, A]\!]_t := \Pi A : \Box_i \times \mathbb{B}.\, \pi_1\ A$$

which is inhabited if and only if $\Pi A : \Box_i .\, A$ is $\Box$

Proposition 17. *The above translation can easily accomodate the presence of an impredicative universe $*$ as long as there is one in the target system, by defining*

$$[*]_t := (* \times \mathbb{B}, \mathtt{true}).$$

The rules for an impredicative universe from Figure 3 are valid with this interpretation.

More generally, the translation can be extended to any type former Φ which is not a universe by defining

$$[\Phi\ (M_1, \ldots, M_n)]_t := (\Phi\ ([M_1]_t, \ldots, [M_n]_t), \mathtt{true}).$$

In the remainder of this section, we assume that we have an equality and an empty type and their translations at hand.

Theorem 18. *We define propositional extensionality as:*

$$\mathtt{propext} := \Pi(A\, B : *).\, (A \to B) \to (B \to A) \to A =_* B.$$

There is in the target system a proof of $[\![\mathtt{propext} \to \bot]\!]_t$.

Proof. Just as in the functional extensionality proof, it is sufficient to take A and B to be the same underlying proposition X, but with a different boolean. $\Box$

Using the validity of propositional extensionality in the set model (Werner 1997), we get the independence of propositional extensionality with CC_ω.

In the same way, it is possible that the negation of univalence holds in the source theory. Using the validity of univalence in the simplicial model of (Kapulkin et al. 2012), we conclude that univalence is independent from CC_ω.

5.2 Ad-hoc polymorphism

In this section, we push the concept of type intensionality into its utmost consequencies, namely by showing that there exists a syntactic model of type theory interpreting a quoting operator on types. Such an operator allows to do case-analysis on the normal form of any type in the theory, which proves that ad-hoc polymorphism is actually compatible with usual type theory. Although this result looks surprising, the model is actually quite straightfoward to obtain as soon as the target theory is expressive enough.

We will assume here that our target theory features induction-recursion, and we will use it to define the type of codes (as in (Dybjer 2000)).

Definition 19 (Codes)**.** We define a family of inductive-recursive definitions $\mathtt{TYPE}_i$ and $\mathtt{Elt}_i$ where i ranges over $\mathbb{N}$ as follows, using a suggestive syntax.

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathtt{Inductive\ TYPE}_0 : \Box_1 :=} \\
\mid \Pi_0^0 & : & \Pi A : \mathtt{TYPE}_0 .\, (\mathtt{Elt}_0\ A \to \mathtt{TYPE}_0) \to \mathtt{TYPE}_0 \\
\multicolumn{3}{l}{\mathtt{with\ Elt}_0 : \mathtt{TYPE}_0 \to \Box_0 := \mathtt{fun}} \\
\mid \Pi_0^0\ A\ B & \Rightarrow & \Pi x : \mathtt{Elt}_0\ A .\, \mathtt{Elt}_0\ (B\ x) \\
\quad \ldots & & \\
\multicolumn{3}{l}{\mathtt{Inductive\ TYPE}_{i+1} : \Box_{i+2} :=} \\
\mid \mathcal{U}_i & : & \mathtt{TYPE}_{i+1} \\
\mid \Pi_{i+1}^0 & : & \Pi A : \mathtt{TYPE}_0 .\, (\mathtt{Elt}_0\ A \to \mathtt{TYPE}_{i+1}) \to \mathtt{TYPE}_{i+1} \\
\mid \ldots & : & \ldots \\
\mid \Pi_{i+1}^{i+1} & : & \Pi A : \mathtt{TYPE}_{i+1} .\, (\mathtt{Elt}_{i+1}\ A \to \mathtt{TYPE}_{i+1}) \to \mathtt{TYPE}_{i+1} \\
\multicolumn{3}{l}{\mathtt{with\ Elt}_{i+1} : \mathtt{TYPE}_{i+1} \to \Box_{i+1} := \mathtt{fun}} \\
\mid \mathcal{U}_i & \Rightarrow & \mathtt{TYPE}_i \\
\mid \Pi_{i+1}^0\ A\ B & \Rightarrow & \Pi x : \mathtt{Elt}_0\ A .\, \mathtt{Elt}_{i+1}\ (B\ x) \\
\mid \ldots & \Rightarrow & \ldots \\
\mid \Pi_{i+1}^{i+1}\ A\ B & \Rightarrow & \Pi x : \mathtt{Elt}_{i+1}\ A .\, \mathtt{Elt}_{i+1}\ (B\ x)
\end{array}$$

The typing and reduction rules generated by these definitions are formally given in Figure 6. These inductive-recursive types respect the usual positivity conditions and thus do not endanger the consistency of the target system.

There are several points to discuss about codes. First, the definition of codes makes a closed-world assumption on the type constructors from the source theory. Furthermore, as codes reflect faithfully the various operations at our disposal in the source theory, we actually need to hardwire the universe hierarchy in the inductive-recursive definitions by externally quantifying over the index $i \in \mathbb{N}$. In order not to make the translation even more complex as it is, we simplify a bit the authorized type formers to prevent a combinatorial explosion. Most notably, the codes for product must be duplicated as many times as there are valid combinations of sorts for it.

It is interesting to remark that the resulting model suffers from an expressivity limitation directly linked to the inductive-recursive construction, that is, it cannot interpret an impredicative universe. This would obviously entail a non-wellfounded loop in the definition of codes, because $* : \Box_0$ but all quantifications targetting $*$ would have to refer to every $\mathtt{TYPE}_i$ defined afterwards.

In order to define the translation, we need to know the universe level at which a term is being typed at each inductive step. Therefore, we will use as a source system CC_ω^ι, a variant of CC_ω which is slightly less expressive but well stratified, defined in Figure 7. It would probably be possible to adapt the translation to the full-blown CC_ω system, at the cost of a more involved translation. Essentially, the main difference is that we annotate both the sequent and the variables from the context with the level of their expected sort. We will omit the indices when they are uniquely defined from the context.

Proposition 20. CC_ω^ι *is a subsystem of* CC_ω.

Proof. It is indeed sufficient to erase all level annotations on colons to translate a well-typed CC_ω^ι term into a well-typed CC_ω term. $\Box$

$$A, B, M, N ::= \ldots \mid \mathtt{TYPE}_i \mid \mathtt{Elt}_i \mid \mathcal{U}_i \mid \mathtt{\Pi}_i^j \mid \mathtt{TYPE\_rec}_i$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathtt{TYPE}_i : \Box_{i+1}} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathtt{Elt}_i : \mathtt{TYPE}_i \to \Box_i} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathcal{U}_i : \mathtt{TYPE}_{i+1}} \qquad \frac{\vdash \Gamma \qquad j \le i}{\Gamma \vdash \mathtt{\Pi}_i^j : \Pi A : \mathtt{TYPE}_j .\, (\mathtt{Elt}_j\ A \to \mathtt{TYPE}_i) \to \mathtt{TYPE}_i}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathtt{TYPE\_rec}_0 : \Pi P : \mathtt{TYPE}_0 \to \Box_j .\, P_{\Pi_0} \to \Pi A : \mathtt{TYPE}_0 .\, P\ A}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathtt{TYPE\_rec}_{i+1} : \Pi P : \mathtt{TYPE}_{i+1} \to \Box_j .\, P\,\mathcal{U}_i \to P_{\Pi^0_{i+1}} \to \ldots \to P_{\Pi^i_{i+1}} \to P_{\Pi_{i+1}} \to \Pi A : \mathtt{TYPE}_{i+1} .\, P\ A}$$

$$\begin{array}{lll}
\text{where} \quad P_{\Pi_i} & := & \Pi (A : \mathtt{TYPE}_i)\,(B : \mathtt{Elt}_i\ A \to \mathtt{TYPE}_i).\, P\ A \to (\Pi x : \mathtt{Elt}_i\ A .\, P\ (B\ x)) \to P\ (\mathtt{\Pi}_i^i\ A\ B) \\
\phantom{\text{where}} \quad P_{\Pi_i^j} & := & \Pi (A : \mathtt{TYPE}_j)\,(B : \mathtt{Elt}_j\ A \to \mathtt{TYPE}_i).\, (\Pi x : \mathtt{Elt}_j\ A .\, P\ (B\ x)) \to P\ (\mathtt{\Pi}_i^j\ A\ B)
\end{array}$$

$$\begin{array}{lll}
\mathtt{Elt}_i\ (\mathtt{\Pi}_i^j\ A\ B) & \equiv & \Pi x : \mathtt{Elt}_j\ A .\, \mathtt{Elt}_i\ (B\ x) \\
\mathtt{Elt}_{i+1}\ \mathcal{U}_i & \equiv & \mathtt{TYPE}_i \\
\mathtt{TYPE\_rec}_0\ P\ p_\Pi\ (\mathtt{\Pi}_0^0\ A\ B) & \equiv & p_\Pi\ (\mathtt{TYPE\_rec}_0\ P\ p_\Pi\ A)\ (\lambda x : \mathtt{Elt}_0\ A .\, \mathtt{TYPE\_rec}_0\ P\ p_\Pi\ (B\ x)) \\
\mathtt{TYPE\_rec}_{i+1}\ P\ p_{\mathcal{U}}\ p_{\Pi_0} \ldots p_{\Pi_{i+1}}\ \mathcal{U}_i & \equiv & p_{\mathcal{U}} \\
\mathtt{TYPE\_rec}_{i+1}\ P\ p_{\mathcal{U}}\ p_{\Pi_0} \ldots p_{\Pi_{i+1}}\ (\mathtt{\Pi}_{i+1}^j\ A\ B) & \equiv & p_{\Pi_j}\ A\ B \\
 & & \quad (\lambda x : \mathtt{Elt}_j\ A .\, \mathtt{TYPE\_rec}_{i+1}\ P\ p_{\mathcal{U}}\ p_{\Pi_0} \ldots p_{\Pi_{i+1}}\ (B\ x)) \\
\mathtt{TYPE\_rec}_{i+1}\ P\ p_{\mathcal{U}}\ p_{\Pi_0} \ldots p_{\Pi_{i+1}}\ (\mathtt{\Pi}_{i+1}^{i+1}\ A\ B) & \equiv & p_{\Pi_{i+1}}\ A\ B\ (\mathtt{TYPE\_rec}_{i+1}\ P\ p_{\mathcal{U}}\ p_{\Pi_0} \ldots p_{\Pi_{i+1}}\ A) \\
 & & \quad (\lambda x : \mathtt{Elt}_{i+1}\ A .\, \mathtt{TYPE\_rec}_{i+1}\ P\ p_{\mathcal{U}}\ p_{\Pi_0} \ldots p_{\Pi_{i+1}}\ (B\ x))
\end{array}$$

Figure 6. Codes

$$\begin{array}{rcl}
\Gamma, \Delta & ::= & \cdot \mid \Gamma, x :_i A \\
A, B, M, N & ::= & \Box_i \mid x \mid M_i\ N \mid \lambda x :_i A .\, M \mid \Pi x :_i A .\, B
\end{array}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \Box_i :_{i+2} \Box_{i+1}} \qquad \frac{\Gamma \vdash A :_{j+1} \Box_j \qquad \Gamma, x :_j A \vdash B :_{i+1} \Box_i \qquad j \le i}{\Gamma \vdash \Pi x :_j A .\, B :_{i+1} \Box_i}$$

$$\frac{\Gamma, x :_i A \vdash M :_j B \qquad \Gamma \vdash \Pi x :_j A .\, B :_{i+1} \Box_i}{\Gamma \vdash \lambda x :_j A .\, M :_i \Pi x :_j A .\, B} \qquad \frac{\Gamma \vdash M :_i \Pi x :_j A .\, B \qquad \Gamma \vdash N :_j A}{\Gamma \vdash M_j\ N :_i B\{x := N\}} \qquad \frac{\Gamma \vdash M :_j B \qquad \Gamma \vdash A :_{i+1} \Box_i}{\Gamma, x :_i A \vdash M :_j B}$$

$$\frac{}{\vdash \cdot} \qquad \frac{\Gamma \vdash A :_{i+1} \Box_i}{\vdash \Gamma, x :_i A} \qquad \frac{\Gamma \vdash A :_{i+1} \Box_i}{\Gamma, x :_i A \vdash x :_i A} \qquad \frac{\Gamma \vdash M :_i B \qquad \Gamma \vdash A :_{i+1} \Box_i \qquad A \equiv B}{\Gamma \vdash M :_i A}$$

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

Figure 7. Typing rules of CC_ω^ι

Contrarily to previous translations, this translation needs to be indexed by the universe level. This is because the code for dependent products is not polymorphic in the universe level, which must thus be inserted explicitly during the translation. Therefore, the index is here to keep track of the current universe level.

Definition 21. The ad-hoc polymorphism translation $[\cdot]_{T_{i \in \mathbb{N}}}$ from CC_ω^ι to $\mathrm{CC}_\omega + \mathtt{TYPE}_{i \in \mathbb{N}}$ is defined by induction on terms as follows.

$$\begin{array}{lcl}
[\Box_i]_{T_j} & := & \mathcal{U}_i \\
[x]_{T_i} & := & x \\
[\lambda x :_j A .\, M]_{T_i} & := & \lambda x : [\![A]\!]_{T_j} .\, [M]_{T_i} \\
[M_j\ N]_{T_i} & := & [M]_{T_i}\ [N]_{T_j} \\
[\Pi x :_j A .\, B]_{T_i} & := & \mathtt{\Pi}_{i-1}^j\ [A]_{T_j}\ (\lambda x : [\![A]\!]_{T_{j-1}} .\, [B]_{T_i}) \\
[\![A]\!]_{T_i} & := & \mathtt{Elt}_i\ [A]_{T_{i+1}} \\
[\![\cdot]\!]_T & := & \cdot \\
[\![\Gamma, x :_i A]\!]_T & := & [\![\Gamma]\!]_T, x : [\![A]\!]_{T_i}
\end{array}$$

In general, Π_i^j may not be defined so that we arbitrarily fix $\Pi_i^j := \Pi_i^0$ whenever $i < j$. Note that the context translation is not indexed, as the corresponding level is hardcoded in each variable from the context.

Proposition 22 (Substitution lemma)**.** *For all terms M, N and any variable x s.t. $\Gamma, x :_i A, \Delta \vdash M :_j B$, we have*

$$[M\{x := N\}]_{T_j} \equiv [M]_{T_j}\{x := [N]_{T_i}\}$$

Proof. By induction on the typing derivation of M. Actually, we need a much weaker condition than typing on M for the theorem to hold, which amounts to checking that variables are used at the correct universe level in the axiom case. $\square$

Theorem 23 (Typing soundness)**.** *If $\Gamma \vdash M :_i A$ then $[\![\Gamma]\!]_T \vdash [M]_{T_i} : [\![A]\!]_{T_i}$.*

Proof. By induction on the typing derivation of M. Conversion is interpreted by conversion itself, thanks to the substitution lemma, and the other rules are staightforward. $\square$

Remark 24. When the source theory only features Π-types and universes, it is not clear whether the translation preserves consistency. Indeed, we have

$$[\![\Pi A :_i \square_i.\, A]\!]_{T_i} \equiv \Pi A : \mathtt{TYPE}_i.\, \mathtt{Elt}_i\ A$$

but then there is no obvious closed code A made out of Π-types and universes to give to a term of the above type in order to recover an inconsistency. For instance, trying to feed it with the code of the impredicative encoding of falsity simply gives back the same type.

This phenomenon does not occur as soon as the source theory features an empty type $\bot$, reflected in the codes as $\mathtt{F}_i$ s.t. $\mathtt{Elt}_i\ \mathtt{F}_i \equiv \bot$. In this case, we can instantiate a proof of the above type with $\mathtt{F}_i$ and we obtain immediately a proof of the empty type.

Theorem 25 (Consistency preservation)**.** *If the source theory features an empty type $\bot$, the ad-hoc polymorphic translation preserves consistency.*

We now turn to show that this translation allows to fully observe the normal form of types by doing case analysis on any term $A : \square_i$. We define formally what we mean by this under the notion of *type quoting* below.

Definition 26 (Type quoting)**.** A type quoting operator is given in CC_ω^ι by the data described in Figure 8.

Theorem 27 (Type Quote)**.** *The $[\cdot]_{T_i}$ translation gives a computational content to type quoting.*

Proof. As all types are represented by codes, the various `quote` operators are simply defined in terms of the `TYPE_rec` operators.

$$[\mathtt{quote}_i\ P\ \vec{M}]_{T_{i+1}} := \\ \mathtt{TYPE\_rec}_i\ (\lambda A : \mathtt{TYPE}_i.\, \mathtt{Elt}_i\ ([P]_{T_{i+2}}\ A))\ [\vec{M}]_{T_{i+1}}$$

Preservation of typing and reduction is straightfoward, as everything has been done to obtain this property. $\square$

The type operator can be used to define non-parametric functions. For example, it becomes possible to prove

$$\Sigma f : (\Pi A : \square.\, A \to A).\, f =_{\Pi A:\square.\, A \to A} \mathtt{id} \to \bot$$

by simply passing a function that is the negation on $\mathbb{B}$ and the identity on other types. This shows that parametricity is not built-in in CC_ω.

6. Formalization and Instrumentation

This section presents the framework for the Coq formalization of some of the translations. We then describe the instrumentation of such translations as a Coq plugin.

6.1 Deep Embedding using De Bruijn Indices

We rely on the formalization of Pure Type Systems (PTS) given by Siles and Herbelin (Siles and Herbelin 2012). It consists in first defining untyped terms and then defining the conversion and typing rules. Note that we could have also used another approach based on induction-recursion that defines directly well-typed terms as in the recent work of Altenkirch and Kaposi (Altenkirch and Kaposi 2016). However, it appears that this approach is a bit too rigid as it forbids to use an untyped term during the translation, making the translation very difficult to define in practice.

Terms are defined by an inductive type, where variables are represented using De Bruijn indices and `Sorts` is an inductive type representing the sorts (either $\square_i$ or $\square_i$ and $*$).

```
Inductive Term : Set :=
 | Var : ℕ → Term
 | Sort : Sorts → Term
 | Π : Term → Term → Term
 | λ : Term → Term → Term
 | App : Term → Term → Term
 | Eq : ∀ (A t₁ t₂ : Term), Term
 | refl : Term → Term
 | J : ∀ (A P t₁ u t₂ p : Term), Term.
```

The substitution of the n-th De Bruijn variable by `N` in `M` is defined by recursion on `M`. We write it `M [n ↦ N]`. Conversion in the calculus, written `M ≡ N`, is defined as the reflexive, symmetric and transitive closure of the beta reduction, defined as an inductive family. The type system is defined by mutual induction with the well-formedness of contexts as

```
Inductive wf : Env → Prop :=
 | wf_nil  : nil ⊣
 | wf_cons : ∀ Γ A s, Γ ⊢ A : Sort s → A::Γ ⊣
where "Γ ⊣" := (wf Γ)
with typ : Env → Term → Term → Prop :=
 | cVar  : ∀ Γ A v, Γ ⊢v → A ↓ v ⊂ Γ → Γ ⊢ Var v : A
```

$$A, B, M, N ::= \ldots \mid \texttt{quote}_0\ P\ N\ A \mid \texttt{quote}_{i+1}\ P\ M\ N_0 \ldots N_{i+1}\ A$$

$$\frac{\Gamma \vdash P : \Box_0 \to \Box_0 \qquad \Gamma \vdash p_{\Pi_0} : P_{\Pi_0} \qquad \Gamma \vdash A : \Box_0}{\Gamma \vdash \texttt{quote}_0\ P\ p_{\Pi_0}\ A : P\ A}$$

$$\frac{\Gamma \vdash P : \Box_{i+1} \to \Box_{i+1} \quad \Gamma \vdash p_{\mathcal{U}} : P\ \Box_i \quad \Gamma \vdash p_{\Pi_0} : P_{\Pi^0_{i+1}} \quad \ldots \quad \Gamma \vdash p_{\Pi_i} : P_{\Pi^i_{i+1}} \quad \Gamma \vdash p_{\Pi_{i+1}} : P_{\Pi_{i+1}} \quad \Gamma \vdash A : \Box_{i+1}}{\Gamma \vdash \texttt{quote}_{i+1}\ P\ p_{\mathcal{U}}\ p_{\Pi_0} \ldots p_{\Pi_{i+1}}\ A : P\ A}$$

$$\begin{array}{lll} \text{where} \quad P_{\Pi_i} & := & \Pi(A : \Box_i)\,(B : A \to \Box_i).\, P\ A \to (\Pi x :_i A.\, P\ (B\ x)) \to P\ (\Pi x :_i A.\, B\ x) \\ \phantom{\text{where}} \quad P_{\Pi^j_i} & := & \Pi(A : \Box_j)\,(B : A \to \Box_i).\, (\Pi x :_j A.\, P\ (B\ x)) \to P\ (\Pi x :_j A.\, B\ x) \end{array}$$

$$\begin{array}{lll} \texttt{quote}_0\ P\ p_{\Pi_0}\ (\Pi^0_0\ A\ B) & \equiv & p_{\Pi_0}\ (\texttt{quote}_0\ P\ p_{\Pi_0}\ A)\ (\lambda x :_0 A.\, \texttt{quote}_0\ P\ p_{\Pi_0}\ (B\ x)) \\ \texttt{quote}_{i+1}\ P\ p_{\mathcal{U}}\ p_{\Pi_0} \ldots p_{\Pi_{i+1}}\ \mathcal{U}_i & \equiv & p_{\mathcal{U}} \\ \texttt{quote}_{i+1}\ P\ p_{\mathcal{U}}\ p_{\Pi_0} \ldots p_{\Pi_{i+1}}\ (\Pi x :_j A.\, B) & \equiv & p_{\Pi_j}\ A\ B \\ & & \quad (\lambda x : A.\, \texttt{quote}_{i+1}\ P\ p_{\mathcal{U}}\ p_{\Pi_0} \ldots p_{\Pi_{i+1}}\ (B\ x)) \\ \texttt{quote}_{i+1}\ P\ p_{\mathcal{U}}\ p_{\Pi_0} \ldots p_{\Pi_{i+1}}\ (\Pi x :_{i+1} A.\, B) & \equiv & p_{\Pi_{i+1}}\ A\ B\ (\texttt{quote}_{i+1}\ P\ p_{\mathcal{U}}\ p_{\Pi_0} \ldots p_{\Pi_{i+1}}\ A) \\ & & \quad (\lambda x : A.\, \texttt{quote}_{i+1}\ P\ p_{\mathcal{U}}\ p_{\Pi_0} \ldots p_{\Pi_{i+1}}\ (B\ x)) \end{array}$$

Figure 8. Type Quoting

```
 | cSort : ∀ Γ s s', Γ ⊣ → Ax s s' → Γ ⊢ Sort s : Sort s'
 | cΠ  : ∀ Γ A B s s' s'', Rel s s' s'' → Γ ⊢ A : Sort s →
                    A::Γ ⊢ B : Sort s' → Γ ⊢ Π A B : Sort s''
 | cλ  : ∀ Γ A B s s' s'' M, Rel s s' s'' → Γ ⊢ A : Sort s →
   A::Γ ⊢ B : Sort s' → A::Γ ⊢ M : B → Γ ⊢ λ A M : Π A B
 | cApp  : ∀ Γ M N A B , Γ ⊢ M : Π A B → Γ ⊢ N : A →
                        Γ ⊢ App M N : B[0 ↦ N]
 | Cnv  : ∀ Γ M A B s, A ≡ B → Γ ⊢ B : Sort s →
           Γ ⊢ M : A → Γ ⊢ M : B
 ... (* typing rules for equality *)
where "Γ ⊢ t : A" := (typ Γ t A).
```

where A ↓ v ⊂ Γ means that the v^th variable has type A in Γ and Ax and Rel are two inductive families reflecting the hierarchy on universes.

It is straightforward to extend Term to integrate other types such as Σ-types, booleans and streams.

Now suppose S is a module defining the PTS for CC_ω (plus dependent pairs and identity type) and T the same PTS extended with booleans. The intensional functions translation introduced in Section 3 can be defined directly by induction on S.Term.

```
Fixpoint tsl (t : S.Term) : T.Term :=
 match t with
  | S.Var v  ⇒ T.Var v
  | S.Sort s  ⇒ T.Sort s
  | S.Π A B  ⇒ T.Σ (Π A^t B^t) Bool
  | S.λ A M  ⇒ T.Pair (λ A^t M^t) true
  | S.App M N  ⇒ T.App (π₁ M^t) N^t
  | S.Eq A t₁ t₂  ⇒ T.Eq A^t t₁^t t₂^t
  | S.refl e  ⇒ T.refl e^t
  | S.J A P t₁ u t₂ p  ⇒ T.J A^t P^t t₁^t u^t t₂^t p^t
 end where "M^t" := (tsl M).
```

The typing soundness of the translation amounts to prove the following theorem by mutual induction.

```
Theorem tsl_correctness :
   (∀ Γ, Γ ⊣ → Γ^t ⊣) ∧ (∀ Γ M A, Γ ⊢ M : A → Γ^t ⊢ M^t : A^t).
```

In the additional materials accompanying this paper[3], we have formalized the intensional functions translation, the intensional streams translation[4] and ad-hoc intensional types translation. In each case, computational soundness, typing soundness and consistency preservation have been checked.

6.2 Instrumentation as a Coq Plugin

It is quite commonly agreed upon that type theory can be difficult to work with formally, which is why proof assistants were implemented in the first place. In particular, it is not very convenient to prove new logical or computational principles directly through the paper translation, let alone in the deep embedding formalization. Luckily, our translations rely on a rather generic target theory, so that we can use the one underlying the Coq proof assistant.

This is where plugins shine in. A Coq plugin is simply a program that, given a Coq proof term M, produces the translation $[M]$ as another Coq term. This is a shallow embedding, and this process is orthogonal to the Coq formalization which uses deep embedding instead. Typically, the fact that $[M]$ is still well-typed relies on the soudness theorem that lives in the metatheory rather than in the object theory.

While there is no real point in implementing model translations in any formal foundations such as e.g. set theory, here, the fact that we have an actual proof assistant at hand makes the plugin quite useful. It allows to give the impression to the user that she is working transparently in the

[3] https://github.com/CoqHott/Program-translations-CC-omega

[4] Some bureaucratic lemmas about lifting of De Bruijn indices have been admitted for the type system extended with streams. See Readme.md.

source theory, while it turns out everything is translated into the target theory on the fly. In particular, typechecking remains decidable by construction, and one can take advantage of all of the facilities provided by the host language. This effectively prevents the necessity to reimplement a standalone proof assistant for the source theory.

Writing a plugin can be cumbersome, for two unrelated reasons. One has to know the internals of the COQ system, which strongly restricts the amount of people able to do it. Furthermore, a distinct term-translating program must be written for every translation one wishes to internalize, which implies that each translation requires a distinct plugin. For instance, the forcing plugin described in (Jaber et al. 2016) is quite different from the ones we implemented for this paper. As of today, there is no such thing as a meta-plugin that would allow to quickly prototype a term translation in COQ.

Remark 28. The type theory implemented by COQ and the various extensions of CC_ω are slightly different, leading to small discrepancies between the previously described translations and their actual instrumentation as a plugin. First, parameters do not appear in dependent pairs from the PTS presentation, while they do in COQ. Second, and more importantly, the management of universes in COQ is more complex as it deals with universe variables and a graph of constraints.

In the remaining of this section, we describe the commands provided by a plugin. First, any COQ term also pertaining to the source theory can be translated into the target theory automatically. Assuming some constant c : A, the plugin command

```
Translate c.
```

produces a new constant $\mathtt{c}^f : [\![A]\!]$ that is added to the current environment and whose body is the translation of the body of c.

Moreover, it is possible to add new symbols to the system through a translation. To do so, it is sufficient to provide for any such symbol $p : A$ its translation $p^\bullet : [\![A]\!]$. This is reflected in the COQ plugin by the command

```
Implement p : A.
```

which opens a new goal of type $[\![A]\!]$. When it is solved, the resulting term $p^\bullet$ is used to automatically extend the translation with a term $p : A$ by defining $[p] := p^\bullet$.

Theoretically, this amounts to considering a typing relation $\vdash_t$ in the translation layer defined by extending CIC with the axiom $\Gamma \vdash_t p : A$. We easily get that if $\Gamma \vdash_t M : B$ then $[\![\Gamma]\!] \vdash [M] : [\![B]\!]$ using the typing correctness of the translation, which justifies the abovementioned equiconsistency result.

In practice, the plugin can thus be used to inhabit new axioms. For instance, for the case of the intensional functions translation, we can add the negation of functional extensionality in the translation layer by providing a term

```
Implement neg_fun_ext :
  (∀ A B, (f g : A → B), (∀ x, f x = g x) → f = g) → False.
```

Such a plugin has been defined for the intensional functions translation, the intensional streams translation and adhoc intensional types translation. Code and examples come with this article.

7. Future Work

As a first step, we wish to improve the plugins which are only prototypes for the moment. This requires generalizing the program transformations presented in this paper to the entire language of COQ—defining it on all inductive types, coinductive types and records. This way, it will be possible to work on the translation layer without restriction and still benefit from the new logical or computional principles available in this layer.

Regarding the type quote operator, an instrumentation as a COQ plugin would require to define induction-recursion in COQ. This is the subject of on-going work and has already been implemented in an experimental branch of COQ 8.5 developed by Matthieu Sozeau (`https://github.com/mattam82/coq/tree/IR`). However, the extension of the syntactic guard condition from inductive to inductive-recursive definitions is still the subject of ongoing research.

Another possible line of work would be on the development of a generic COQ plugin that allows users to define their own compilation phase with a minimum effort—e.g., by defining only the translation on untyped terms in the deep embedding—and generate automatically the associated plugin for this transformation. This way, it will be possible to experiment the frontier of what is provable in type theory at a low cost investment.

But the most important is the development of more translations in order to realize axioms and logic principles. For instance, we wonder if it is possible to find a presentation of the setoid model as a program translation.

References

T. Altenkirch. Extensional equality in intensional type theory. In *Proceedings of LICS*, Trento, Italy, July 1999.

T. Altenkirch and A. Kaposi. Type theory in type theory using quotient inductive types. In *Proceedings of POPL*, 2016.

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.

J.-P. Bernardy and G. Moulin. A computational interpretation of parametricity. In *Proceedings of LICS*, Dubrovnik, Croatia, June 2012.

J.-P. Bernardy, P. Jansson, and R. Paterson. Parametricity and dependent types. In *Proceedings of ICFP*, Baltimore, Maryland, USA, 2010.

J. Chapman. Type theory should eat itself. *Electron. Notes Theor. Comput. Sci.*, 228:21–36, Jan. 2009. ISSN 1571-0661.

P. Dybjer. Internal type theory. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs*, volume 1158 of *LNCS*, pages 120–134. Springer Berlin Heidelberg, 1996. ISBN 978-3-540-61780-8.

P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *J. Symb. Log.*, 65(2):525–549, 2000.

M. Hoffman. Syntax and semantics of dependent types. In *Semantics and Logics of Computation*, pages 241–298, 1997.

M. Hofmann. *Extensional constructs in intensional type theory*. CPHC/BCS distinguished dissertations. Springer, 1997. ISBN 978-3-540-76121-1.

G. Jaber, N. Tabareau, and M. Sozeau. Extending Type Theory with Forcing. In *Proceedings of LICS*, Dubrovnik, Croatia, June 2012.

G. Jaber, G. Lewertoski, P.-M. Pédrot, N. Tabareau, and M. Sozeau. The Definitional Side of the Forcing. In *Proceedings of LICS*, New-York, USA, July 2016.

B. Jacobs. Comprehension categories and the semantics of type dependency. *Theoretical Computer Science*, 107(2):169 – 207, 1993.

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

Y. Lafont, B. Reus, and T. Streicher. *Continuation semantics or expressing implication by negation*. Univ. München, Inst. für Informatik, 1993.

Z. Luo. ECC, an extended calculus of constructions. In *Proceedings of LICS*, Pacific Grove, CA, USA, June 1989.

C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. Thèse d'université, Paris 7, Jan. 1989. URL `http://www.lri.fr/~paulin/PUBLIS/these.ps.gz`.

V. Siles and H. Herbelin. Pure type systems conversion is always typable. *Journal of Functional Programming*, 22(2):153–180, Mar 2012.

The AGDA Development Team. *Agda*. 2015. URL `http://wiki.portal.chalmers.se/agda`.

The COQ Development Team. *The Coq proof assistant reference manual*. 2015. URL `http://coq.inria.fr`. Version 8.5.

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