In Memory of Martin Hofmann

# Eliminating Reflection from Type Theory[*]

Théo Winterhalter
Gallinette Project-Team, Inria
Nantes, France
theo.winterhalter@inria.fr

Matthieu Sozeau
Pi.R2 Project-Team, Inria and IRIF
Paris, France
matthieu.sozeau@inria.fr

Nicolas Tabareau
Gallinette Project-Team, Inria
Nantes, France
nicolas.tabareau@inria.fr

## Abstract

Type theories with equality reflection, such as extensional type theory (ETT), are convenient theories in which to formalise mathematics, as they make it possible to consider provably equal terms as convertible. Although type-checking is undecidable in this context, variants of ETT have been implemented, for example in NuPRL and more recently in Andromeda. The actual objects that can be checked are not proof-terms, but derivations of proof-terms. This suggests that any derivation of ETT can be translated into a typecheckable proof term of intensional type theory (ITT). However, this result, investigated categorically by Hofmann in 1995, and 10 years later more syntactically by Oury, has never given rise to an effective translation. In this paper, we provide the first effective syntactical translation from ETT to ITT with uniqueness of identity proofs and functional extensionality. This translation has been defined and proven correct in Coq and yields an executable plugin that translates a derivation in ETT into an actual Coq typing judgment. Additionally, we show how this result is extended in the context of homotopy type theory to a two-level type theory.

***CCS Concepts*** • **Theory of computation** → **Proof theory**; **Type theory**; *Constructive mathematics*; Logic and verification; Automated reasoning;

***Keywords*** dependent types, translation, formalisation

**ACM Reference Format:**
Théo Winterhalter, Matthieu Sozeau, and Nicolas Tabareau. 2019. Eliminating Reflection from Type Theory. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '19), January 14–15, 2019, Cascais, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3293880.3294095

[*]This work is supported by the CoqHoTT ERC Grant 64399.

## 1 Introduction

Type theories with equality reflection, such as extensional type theory (ETT), are convenient theories in which to formalise mathematics, as they make it possible to consider provably equal terms as convertible, as expressed in the following typing rule:

$$\frac{\Gamma \vdash_{\mathsf{x}} e : u =_A v}{\Gamma \vdash_{\mathsf{x}} u \equiv v : A} \tag{1}$$

Here, the type $u =_A v$ is Martin-Löf's identity type with only one constructor $\mathsf{refl}\ u : u =_A u$ which represents proofs of equality inside type theory, whereas $u \equiv v : A$ means that $u$ and $v$ are convertible in the theory—and can thus be silently replaced by one another in any term. Several variants of ETT have been considered and implemented, for example in NuPRL[2] [Allen et al. 2000] and more recently in Andromeda [Bauer et al. 2016]. The prototypical example of the use of equality reflection is the definition of a coercion function between two types $A$ and $B$ that are equal (but not convertible) by taking a term of type $A$ and simply returning it as a term of type $B$:

$$\lambda\, A\, B\, (e : A = B)\, (x : A).\, x : \Pi\, A\, B.\, A = B \to A \to B.$$

In intensional type theory (ITT), this term does not typecheck because $x$ of type $A$ can not be given the type $B$ by conversion. In ETT, however, equality reflection can be used to turn the witness of equality into a proof of conversion and thus the type system validates the fact that $x$ can be given the type $B$. This means that one needs to guess equality proofs during type-checking, because the witness of equality has been lost at the application of the reflection rule. Guessing it was not so hard in this example but is in general undecidable, as one can for instance encode the halting problem of any Turing machine as an equality in ETT. That is, the actual objects that can be checked in ETT are not terms, but instead derivations of terms. It thus seems natural to wonder whether any derivation of ETT can be translated into a typecheckable term of ITT. And indeed, it is well know that one can find a corresponding term of the same type in ITT by *explicitly* transporting the term $x$ of type $A$ using the elimination of internal equality on the witness of equality $e$, noted $e_*$:

$$\lambda\, A\, B\, (e : A = B)\, (x : A).\, e_*\, x : \Pi\, A\, B.\, A = B \to A \to B.$$

[2]Although the reflection rule is provable in NuPRL, its calculus is based on realisability rather than on intensional type theory plus reflection.

This can be seen as a way to make explicit the *silent* use of reflection. Furthermore, by making the use of transport as economic as possible, the corresponding ITT term can be seen as a compact witness of the derivation tree of the original ETT term.

This result has first been investigated categorically in the pioneering work of Hofmann [1995, 1997], by showing that the term model of ITT can be turned into a model of ETT by quotienting this model with propositional equality. However, it is not clear how to extend this categorical construction to an explicit and constructive translation from a derivation in ETT to a term of ITT. In 2005, this result has been investigated more syntactically by Oury [2005]. However, his presentation does not give rise to an effective translation. By an *effective* translation we mean that it is entirely constructive and can be used to deterministically *compute* the translation of a given ETT typing derivation. Two issues prevent deriving an *effective* translation from Oury's presentation, and it is the process of actual formalisation of the result in a proof assistant that led us to these discoveries. First, his handling of related contexts is not explicit enough, which we fix by framing the translation using ideas coming from the parametricity translation (Section 1.2). Additionally, Oury's proof requires an additional axiom in ITT on top of functional extensionality (FunExt) and uniqueness of identity proofs, that has no clear motivation and can be avoided by considering an annotated syntax (Section 2.1).

***Contributions.*** In this paper, we present the first effective syntactical translation from ETT to ITT (assuming uniqueness of identity proofs (UIP) and FunExt in ITT). By syntactical translation, we mean an explicit translation from a derivation $\Gamma \vdash_{\mathsf{x}} t : T$ of ETT (the x index testifies that it is a derivation in ETT) to a context $\Gamma'$, term $t'$ and type $T'$ of ITT such that $\Gamma' \vdash t' : T'$ in ITT. This translation enjoys the additional property that if $T$ can be typed in ITT, *i.e.,* $\Gamma \vdash T$, then $T' \equiv T$. This means in particular that a theorem proven in ETT but whose statement is also valid in ITT can be automatically transferred to a theorem of ITT. For instance, one could use a *local* extension of the Coq proof assistant with a reflection rule, without being forced to rely on the reflection in the entire development.

This translation can be seen as a way to build a syntactical model of ETT from a model of ITT as described more generally in Boulier et al. [2017] and has been entirely programmed and formalised in Coq [Coq development team 2017]. For this, we rely on TemplateCoq[3] [Anand et al. 2018], which provides a reifier for Coq terms as represented in Coq's kernel as well as a formalisation of the type system of Coq. Thus, our formalisation of ETT is just given by adding the reflection rule to a subset of the original type system of Coq. This allows us to extract concrete Coq terms and types from a closed derivation of ETT, using a little trick to incorporate Inductive types and induction. We do not treat cumulativity of universes which is an orthogonal feature of Coq's type theory. It would also complicate the proof which relies on uniqueness of typing.

[3]https://metacoq.github.io/metacoq/

***Outline of the Paper.*** Before going into the technical development of the translation, we explain its main ingredients and differences with previous works. Then, in Section 2, we define the extensional and intensional type theories we consider. In Section 3, we define the main ingredient of the translation, which is a relation between terms of ETT and terms in ITT. Then, the translation is given in Section 4. Section 5 describes the Coq formalisation and Sections 6 and 7 discuss limitations and related work. The details can be found in the long version: https://hal.archives-ouvertes.fr/hal-01849166

The Coq formalisation can be found in https://github.com/TheoWinterhalter/ett-to-itt.

## 1.1 On the Need for UIP and FunExt

Our translation targets ITT plus UIP and FunExt, which correspond to the two following axioms (where $\square_i$ denotes the universe of types at level $i$):

$$\begin{aligned}\mathtt{UIP} &: \Pi(A : \square_i)\ (x\ y : A)\ (e\ e' : x = y).\ e = e' \\ \mathtt{FunExt} &: \Pi(A : \square_i)\ (B : A \to \square_i)\ (f\ g : \Pi(x : A).\ B\ x). \\ &\quad (\Pi(x : A).\ f\ x = g\ x) \to f = g\end{aligned}$$

The first axiom says that any two proofs of the same equality are equal, and the other one says that two (dependent) functions are equal whenever they are pointwise equal[4]. These two axioms are perfectly valid statements of ITT and they can be proven in ETT. Indeed, UIP can be shown to be equivalent to the Streicher's axiom K

$$\mathtt{K} \quad : \quad \Pi(A : \square_i).\ \Pi(x : A).\ \Pi(e : x = x).\ e = \mathsf{refl}_x$$

using the elimination on the identity type. But K is provable in ETT by considering the type

$$\Pi(A : \square_i).\ \Pi(x\ y : A).\ \Pi(e : x = y).\ e = \mathsf{refl}_x$$

which is well typed (using the reflection rule to show that $e$ has type $x = x$) and which can be inhabited by elimination of the identity type. In the same way, FunExt is provable in ETT because

$$\begin{array}{lll} & \Pi(x : A).\ f\ x = g\ x & \\ \to & x : A \vdash f\ x \equiv g\ x & \text{by reflection} \\ \to & (\lambda(x : A).f\ x) \equiv (\lambda(x : A).g\ x) & \text{by congruence of } \equiv \\ \to & f \equiv g & \text{by } \eta\text{-law} \\ \to & f = g & \end{array}$$

Therefore, applying our translation to the proofs of those theorems in ETT gives corresponding proofs of the same theorems in ITT. However, UIP is independent from ITT, as first shown by Hofmann and Streicher using the groupoid

[4]In Homotopy Type Theory (HoTT) [Univalent Foundations Program 2013], FunExt is stated in a more complete way, using the notion of adjoint equivalences, but this more complete way collapses to our simpler statement in presence of UIP.

model [Hofmann and Streicher 1998], which has recently been extended in the setting of univalent type theory using the simplicial or cubical models [Bezem et al. 2013; Kapulkin and Lumsdaine 2012]. Similarly, FunExt is independent from ITT, it is folklore but has recently been formalised by Boulier *et al.* using a simple syntactical translation [Boulier et al. 2017].

Therefore, our translation provides proofs of axioms independent from ITT, which means that the target of the translation already needs to have both UIP and FunExt. These last two elements are only necessary in ITT and could be removed from ETT but this allows us to consider only one syntax for both. Part of our work is to show formally that they are the only axioms required.

### 1.2 Heterogeneous Equality and the Parametricity Translation

The basic idea behind the translation from ETT to ITT is to interpret conversion using the internal notion of equality, *i.e.,* the identity type. But this means that two terms of two convertible types that were comparable in ETT become comparable in ITT only up-to the equality between the two types. One possible solution to this problem is to consider a native heterogeneous equality, such as *John Major equality* introduced by McBride [2000]. However, to avoid adding additional axioms to ITT as done by Oury [2005], we prefer to encode this heterogeneous equality using the following dependent sums:

$$t \;{}_{T}{\cong}_{U}\; u := \Sigma(p : T = U).\, p_* \; t = u.$$

During the translation, the same term occurring twice can be translated in two different manners, if the corresponding typing derivations are different. Even the types of the two different translations may be different. However, we have the strong property that any two translations of the same term only differ in places where transports of proof of equality have been injected. To keep track of this property, we introduce the relation $t \sim t'$ between two terms of ITT, of possibly different types. The crux of the proof of the translation is to guarantee that for every two terms $t_1$ and $t_2$ such that $\Gamma \vdash t_1 : T_1, \Gamma \vdash t_2 : T_2$ and $t_1 \sim t_2$, there exists $p$ such that $\Gamma \vdash p : t_1 \;{}_{T_1}{\cong}_{T_2}\; t_2$. However, during the proof, variables of different but (propositionally) equal types are introduced and the context cannot be maintained to be the same for both $t_1$ and $t_2$. Therefore, the translation needs to keep track of this duplication of variables, plus a proof that they are heterogeneously equal. This mechanism is similar to what happens in the (relational) internal parametricity translation in ITT introduced by Bernardy et al. [2012] and recently rephrased in the setting of TemplateCoq [Anand et al. 2018]. Namely, a context is not translated as a telescope of variables, but as a telescope of triples consisting of two variables plus a witness that they are in the parametric relation. In our setting, this amounts to consider telescope of triples consisting of two variables plus a witness that they are heterogeneously equal. We can express this by considering the following dependent sums:

$$\text{Pack } A_1 \; A_2 := \Sigma(x : A_1).\, \Sigma(y : A_2).\, x \;{}_{A_1}{\cong}_{A_2}\; y.$$

This presentation inspired by the parametricity translation is crucial in order to get an effective translation, because it is necessary to keep track of the evolution of contexts when doing the translation on open terms. This ingredient is missing in Oury's work [Oury 2005], which prevents him from deducing an effective (*i.e.,* constructive and computable) translation from his theorem.

## 2 Definitions of Extensional and Intensional Type Theories

This section presents the common syntax, typing and main properties of ETT and ITT. Our type theories feature a universe hierarchy, dependent products and sums as well as Martin Löf's identity types.

### 2.1 Syntax of ETT and ITT

The common syntax of ETT and ITT is given in Figure 1. It features: dependent products $\Pi(x : A).\; B$, with (annotated) $\lambda$-abstractions and (annotated) applications, negative dependent sums $\Sigma(x : A).\, B$ with (annotated) projections, sorts $\square_i$, identity types $u =_A v$ with reflection and elimination as well as terms realising UIP and FunExt. Annotating terms with otherwise computationally irrelevant typing information is a common practice when studying the syntax of type theory precisely (see [Streicher 1993] for a similar example). We will write $A \to B$ for $\Pi(\_ : A).\; B$ the non-dependent product / function type.

We consider a fixed universe hierarchy without cumulativity, which ensures in particular uniqueness of typing (2.2) which is important for the translation.

***About Annotations.*** Although it may look like a technical detail, the use of annotation is more fundamental in ETT than it is in ITT (where it is irrelevant and doesn't affect the theory). And this is actually one of the main differences between our work (and that of Martin Hofmann [1995] who has a similar presentation) and the work of Oury [2005].

Indeed, by using the standard model where types are interpreted as cardinals rather than sets, it is possible to see that the equality $\text{nat} \to \text{nat} = \text{nat} \to \text{bool}$ is independent from the theory, it is thus possible to assume it (as an axiom, or for those that would still not be convinced, simply under a $\lambda$ that would introduce this equality). In that context, the identity map $\lambda(x : \text{nat}).\; x$ can be given the type $\text{nat} \to \text{bool}$ and we thus type $(\lambda(x : \text{nat}).\; x)\; 0 : \text{bool}$. Moreover, the $\beta$-reduction of the non-annotated system used by Oury concludes that this expression reduces to 0, but cannot be given the type bool (as we said, the equality $\text{nat} \to \text{nat} = \text{nat} \to \text{bool}$ is independent from the theory, so the context is consistent).

| | | | |
|---|---|---|---|
| $s$ | $::=$ | $\square_i \ (i \in \mathbb{N})$ | sorts (universes) |
| $T, A, B, t, u, v$ | $::=$ | $x \mid \lambda(x:A).B.t \mid t \ @_{x:A.B}\, u \mid \Pi(x:A).\ B \mid s$ | dependent $\lambda$-calculus |
| | $\mid$ | $\langle u; v\rangle_{x:A.B} \mid \pi_1^{x:A.B}\, p \mid \pi_2^{x:A.B}\, p \mid \Sigma(x:A).\,B$ | dependent pairs |
| | $\mid$ | $\mathsf{refl}_A\, u \mid \mathsf{J}(A, u, x.e.P, w, v, p) \mid u =_A v$ | propositional equality |
| | $\mid$ | $\mathsf{funext}(x:A, B, f, g, e) \mid \mathsf{uip}(A, u, v, p, q)$ | equality axioms |
| $\Gamma, \Delta$ | $::=$ | $\bullet \mid \Gamma, x:A$ | contexts |

**Figure 1.** Common syntax of ETT and ITT

This means we lack subject reduction in this case (or uniqueness of types, depending on how we see the issue). Our presentation has a blocked $\beta$-reduction limited to matching annotations: $(\lambda(x:A).B.\ t)\ @_{x:A.B}\, u = t[x \leftarrow u]$, from which subject reduction and uniqueness of types follow.

Although subtle, this difference is responsible for Oury's need for an extra axiom. Indeed, to treat the case of equality of applications in his proof, he needs to assume the congruence rule for heterogeneous equality of applications, which is not provable when formulated with John Major equality (Fig. 2). Thanks to annotations and our notion of heterogeneous equality, we can prove this congruence rule for applications.

$$\textsc{JMApp}\quad \frac{f_1 \ {}_{\forall(x:U_1).V_1}{\cong}_{\forall(x:U_2).V_2}\ f_2 \qquad u_1 \ {}_{U_1}{\cong}_{U_2}\ u_2}{f_1\, u_1 \ {}_{V_1[x\leftarrow u_1]}{\cong}_{V_2[x\leftarrow u_2]}\ f_2\, u_2}$$

**Figure 2.** Congruence of heterogeneous equality

## 2.2 The Typing Systems

As usual in dependent type theory, we consider contexts which are telescopes whose declarations may depend on any variable already introduced. We note $\Gamma \vdash t : A$ to say that $t$ has type $A$ in context $\Gamma$. $\Gamma \vdash A$ shall stand for $\Gamma \vdash A : s$ for some sort $s$ and similarly $\Gamma \vdash A \equiv B$ stands for $\Gamma \vdash A \equiv B : s$.

We use two relations $(s, s') \in \mathsf{Ax}$ (written $(s, s')$ for short) and $(s, s', s'') \in \mathsf{R}$ (written $(s, s', s'')$) to constrain the sorts in the typing rules for universes, dependent products and dependent sums, as is done in any Pure Type System (PTS). In our case, because we do not have cumulativity, the rules are as follows:

$$(\square_i, \square_{i+1}) \in \mathsf{Ax} \qquad (\square_i, \square_j, \square_{\max(i,j)}) \in \mathsf{R}$$

We give the typing rules of ITT in Figure 3. The rules are standard and we do not explain them. Let us just point out the conversion rule, which says that $u : A$ can be given the type $u : B$ when $A \equiv B$, *i.e.,* when $A$ and $B$ are convertible. As the notion of conversion is central in our work—the conversion of ETT being translated to an equality in ITT—we provide an exhaustive definition of it, with computational conversion rules (including $\beta$-conversion or reduction of the elimination principle of equality over reflexivity, see Figure 4), however congruence conversion rules are omitted due to lack of space. Note that we use Christine Paulin-Möhring's variant of the J rule rather than Martin-Löf's original formulation. Although pretty straightforward, being precise here is very important, as for instance the congruence rule for $\lambda$-terms is the reason why FunExt is derivable in ETT. Congruence of equality terms is a standard extension of congruence to the new principles we add (UIP and FunExt).

ETT is thus simply an extension of ITT (we write $\vdash_x$ for the associated typing judgment) with the reflection rule on equality, which axiomatises that propositionally equal terms are convertible (see Equation 1). Note that, as already mentioned, in the presence of reflection and J, UIP is derivable so we could remove it from ETT, but keeping it allows us to share a common syntax which makes the statements of theorems simpler and does not affect the development.

## 2.3 General Properties of ITT and ETT

We now state the main properties of both ITT and ETT. We do not detail their proof as they are standard and can be found in the Coq formalisation.

First, although not explicit in the typing system, weakening is admissible in ETT and ITT.

**Lemma 2.1** (Weakening)**.** *If* $\Gamma \vdash \mathcal{J}$ *and* $\Delta$ *extends* $\Gamma$ *(possibly interleaving variables) then* $\Delta \vdash \mathcal{J}$*.*

Then, as mentioned above, the use of a non-cumulative hierarchy allows us to prove that a term $t$ can be given at most one type in a context $\Gamma$, up-to conversion.

**Lemma 2.2** (Uniqueness of typing)**.** *If* $\Gamma \vdash u : T_1$ *and* $\Gamma \vdash u : T_2$ *then* $\Gamma \vdash T_1 \equiv T_2$*.*

Finally, an important property of the typing system (seen as a mutual inductive definition) is the possibility to deduce hypotheses from their conclusion, thanks to inversion of typing. Note that it is important here that our syntax is annotated for applications and projections as it provides a richer inversion principle.

**Lemma 2.3** (Inversion of typing)**.**

1. *If* $\Gamma \vdash x : T$ *then* $(x : A) \in \Gamma$ *and* $\Gamma \vdash A \equiv T$*.*
2. *If* $\Gamma \vdash \square_i : T$ *then* $\Gamma \vdash \square_{i+1} \equiv T$*.*
3. *If* $\Gamma \vdash \Pi(x:A).\ B : T$ *then* $\Gamma \vdash A : s$ *and* $\Gamma, x : A \vdash B : s'$ *and* $\Gamma \vdash s'' \equiv T$ *for some* $(s, s', s'')$*.*

***Well-formedness of contexts.***

$$\frac{}{\vdash \bullet} \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash A}{\vdash \Gamma, x : A}(x \notin \Gamma)$$

***Types.***

$$\frac{\vdash \Gamma}{\Gamma \vdash s : s'}(s, s') \qquad \frac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash B : s'}{\Gamma \vdash \Pi(x : A).\ B : s''}(s, s', s'')$$

$$\frac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash B : s'}{\Gamma \vdash \Sigma(x : A).\, B : s''}(s, s', s'')$$

$$\frac{\Gamma \vdash A : s \qquad \Gamma \vdash u : A \qquad \Gamma \vdash v : A}{\Gamma \vdash u =_A v : s}$$

***Structural rules.***

$$\frac{\vdash \Gamma \qquad (x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash u : A \qquad \Gamma \vdash A \equiv B : s}{\Gamma \vdash u : B}$$

***λ-calculus terms.***

$$\frac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash B : s' \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda(x : A).B.t : \Pi(x : A).\ B}$$

$$\frac{\begin{array}{c}\Gamma \vdash A : s \\ \Gamma, x : A \vdash B : s' \qquad \Gamma \vdash t : \Pi(x : A).\ B \qquad \Gamma \vdash u : A\end{array}}{\Gamma \vdash t \ @_{x:A.B}\ u : B[x \leftarrow u]}$$

$$\frac{\begin{array}{c}\Gamma \vdash u : A \\ \Gamma \vdash A : s \qquad \Gamma, x : A \vdash B : s' \qquad \Gamma \vdash v : B[x \leftarrow u]\end{array}}{\Gamma \vdash \langle u; v \rangle_{x:A.B} : \Sigma(x : A).\, B}$$

$$\frac{\Gamma \vdash p : \Sigma(x : A).\, B}{\Gamma \vdash \pi_1^{x:A.B}\ p : A} \qquad \frac{\Gamma \vdash p : \Sigma(x : A).\, B}{\Gamma \vdash \pi_2^{x:A.B}\ p : B[x \leftarrow \pi_1^{x:A.B}\ p]}$$

***Equality terms.***

$$\frac{\Gamma \vdash A : s \qquad \Gamma \vdash u : A}{\Gamma \vdash \mathsf{refl}_A\ u : u =_A u} \qquad \frac{\Gamma \vdash e_1, e_2 : u =_A v}{\Gamma \vdash \mathsf{uip}(A, u, v, e_1, e_2) : e_1 = e_2}$$

$$\frac{\begin{array}{c}\Gamma \vdash A : s \qquad \Gamma \vdash u, v : A \qquad \Gamma, x : A, e : u =_A x \vdash P : s' \\ \Gamma \vdash p : u =_A v \qquad \Gamma \vdash w : P[x \leftarrow u, e \leftarrow \mathsf{refl}_A\ u]\end{array}}{\Gamma \vdash \mathsf{J}(A, u, x.e.P, w, v, p) : P[x \leftarrow v, e \leftarrow p]}$$

$$\frac{\begin{array}{c}\Gamma \vdash f, g : \Pi(x : A).\ B \\ \Gamma \vdash e : \Pi(x : A).\ f \ @_{x:A.B}\ x =_B g \ @_{x:A.B}\ x\end{array}}{\Gamma \vdash \mathsf{funext}(x : A, B, f, g, e) : f = g}$$

**Figure 3.** Typing rules

4. *If* $\Gamma \vdash \lambda(x : A).B.t : T$ *then* $\Gamma \vdash A : s$ *and* $\Gamma, x : A \vdash B : s'$ *and* $\Gamma, x : A \vdash t : B$ *and* $\Gamma \vdash \Pi(x : A).\ B \equiv T$.
5. *If* $\Gamma \vdash u \ @_{x:A.B}\ v : T$ *then* $\Gamma \vdash A : s$ *and* $\Gamma, x : A \vdash B : s'$ *and* $\Gamma \vdash u : \Pi(x : A).\ B$ *and* $\Gamma \vdash v : A$ *and* $\Gamma \vdash B[x \leftarrow u] \equiv T$.
6. *... Analogous for the remaining term and type constructors.*

*Proof.* Each case is proven by induction on the derivation (which corresponds to any number of applications of the conversion rule following one introduction rule). □

## 3 Relating Translated Expressions

We want to define a relation on terms that equates two terms that are the same up to transport. This begs the question of what notion of transport is going to be used. Transport can be defined from elimination of equality as follows:

**Definition 3.1** (Transport)**.** Given $\Gamma \vdash p : T_1 =_s T_2$ and $\Gamma \vdash t : T_1$ we define the transport of $t$ along $p$, written $p_*\ t$, as $\mathsf{J}(s, T_1, X.e.\ T_1 \to X, \lambda(x : T_1).T_1.x, T_2, p) \ @_{T_1.T_2}\ t$ such that $\Gamma \vdash p_*\ t : T_2$.

However, in order not to confuse the transports added by the translation with the transports that were already present in the source, we consider $p_*$ as part of the syntax in the reasoning. It will be unfolded to its definition only after the complete translation is performed. This idea is not novel as Hofmann already had a Subst operator that was part of his ITT (noted $\mathrm{TT_I}$ in his paper [Hofmann 1995]).

We first define the (purely syntactic) relation $\sqsubset$ between ETT terms and ITT terms in Figure 5 stating that the ITT term is simply a decoration of the first term by transports. Its purpose is to state how close to the original term its translation is. Then, we extend this relation to a similarity relation $\sim$ on ETT terms by taking its symmetric and transitive closure:

$$\sim := (\sqsubset \cup \sqsubset^{-1})^+$$

**Lemma 3.2** ($\sim$ is an equivalence relation)**.** *$\sim$ is reflexive, symmetric and transitive.*

*Proof.* For reflexivity we proceed by induction on the term. □

The goal is to prove that two terms in this relation, that are well-typed in the target type theory, are heterogeneously equal. As for this notion, we recall the definition we previously gave: $t \ {}_T{\cong}_U\ u := \Sigma(p : T = U).\, p_*\ t = u$. This definition of heterogeneous equality can be shown to be reflexive, symmetric and transitive. Because of UIP, heterogeneous equality collapses to equality when taken on the same type.

**Lemma 3.3.** *If* $\Gamma \vdash e : u \ {}_A{\cong}_A\ v$ *then there exists* $p$ *such that* $\Gamma \vdash p : u =_A v$.

*Proof.* This holds thanks to UIP on equality, which implies K, and so the proof of $A = A$ can be taken to be reflexivity. □

***Computation.***

$$\frac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash B : s' \qquad \Gamma, x : A \vdash t : B \qquad \Gamma \vdash u : A}{\Gamma \vdash (\lambda(x : A).B.t) \mathbin{@_{x:A.B}} u \equiv t[x \leftarrow u] : B[x \leftarrow u]}$$

$$\frac{\Gamma \vdash A : s \qquad \Gamma \vdash u : A \qquad \Gamma, x : A, e : u =_A x \vdash P : s' \qquad \Gamma \vdash w : P[x \leftarrow u, e \leftarrow \mathsf{refl}_A\ u]}{\Gamma \vdash \mathsf{J}(A, u, x.e.P, w, u, \mathsf{refl}_A\ u) \equiv w : P[x \leftarrow u, e \leftarrow \mathsf{refl}_A\ u]}$$

$$\frac{\Gamma \vdash A : s \qquad \Gamma \vdash u : A \qquad \Gamma, x : A \vdash B : s' \qquad \Gamma \vdash v : B[x \leftarrow u]}{\Gamma \vdash \pi_1^{x:A.B}\ \langle u; v\rangle_{x:A.B} \equiv u : A}$$

$$\frac{\Gamma \vdash A : s \qquad \Gamma \vdash u : A \qquad \Gamma, x : A \vdash B : s' \qquad \Gamma \vdash v : B[x \leftarrow u]}{\Gamma \vdash \pi_2^{x:A.B}\ \langle u; v\rangle_{x:A.B} \equiv v : B[x \leftarrow u]}$$

***Conversion.***

$$\frac{\Gamma \vdash t_1 \equiv t_2 : T_1 \qquad \Gamma \vdash T_1 \equiv T_2}{\Gamma \vdash t_1 \equiv t_2 : T_2}$$

**Figure 4.** Main conversion rules (omitting congruence rules)

**Note.** *As a corollary, $\cong$ on types corresponds to equality. Indeed when we have $\Gamma \vdash e : A\ {}_s{\cong}_{s'}\ B$ we have that $s = s'$, which implies that $s$ and $s'$ have the same sort and thus are syntactically the same (by an inversion argument).*

Before we can prove the fundamental lemma stating that two terms in relation are heterogeneously equal, we need to consider another construction. As explained in the introduction, when proving the property by induction on terms, we introduce variables in the context that are equal only up-to heterogeneous equality. This phenomenon is similar to what happens in the parametricity translation [Bernardy et al. 2012]. Our fundamental lemma on the decoration relation $\sim$ assumes two related terms of potentially different types $T1$ and $T2$ to produce an heterogeneous equality between them. For induction to go through under binders (e.g. for dependent products and abstractions), we hence need to consider the two terms under different, but heterogeneously equal contexts. Therefore, the context we produce will not only be a telescope of variables, but rather a telescope of triples consisting of two variables of possibly different types, and a witness that they are heterogeneously equal. To make this precise, we define the following macro:

$$\mathsf{Pack}\ A_1\ A_2 := \Sigma(x : A_1).\,\Sigma(y : A_2).\,x \cong y$$

together with its projections

$$\mathsf{Proj}_1\ p := \pi_1^{\cdot}\ p \qquad \mathsf{Proj}_2\ p := \pi_1^{\cdot}\ \pi_2^{\cdot}\ p \qquad \mathsf{Proj}_e\ p := \pi_2^{\cdot}\ \pi_2^{\cdot}\ p.$$

We can then extend this notion canonically to contexts of the same length that are well formed using the same sorts:

$$\begin{aligned}&\mathsf{Pack}\ (\Gamma_1, x : A_1)\ (\Gamma_2, x : A_2) := \\ &(\mathsf{Pack}\ \Gamma_1\ \Gamma_2), x : \mathsf{Pack}\ (A_1[\gamma_1])\ (A_2[\gamma_2])\end{aligned}$$

$$\mathsf{Pack}\ \bullet\ \bullet := \bullet.$$

When we pack contexts, we also need to apply the correct projections for the types in that context to still make sense. Assuming two contexts $\Gamma_1$ and $\Gamma_2$ of the same length, we can define left and right substitutions:

$$\begin{aligned}\gamma_1 &:= [x \leftarrow \mathsf{Proj}_1\ x \mid (x : \_) \in \Gamma_1] \\ \gamma_2 &:= [x \leftarrow \mathsf{Proj}_2\ x \mid (x : \_) \in \Gamma_2].\end{aligned}$$

These substitutions implement lifting of terms to packed contexts: $\Gamma, \mathsf{Pack}\ \Gamma_1\ \Gamma_2 \vdash t[\gamma_1] : A[\gamma_1]$ whenever $\Gamma, \Gamma_1 \vdash t : A$ (resp. $\Gamma, \mathsf{Pack}\ \Gamma_1\ \Gamma_2 \vdash t[\gamma_2] : A[\gamma_2]$ whenever $\Gamma, \Gamma_2 \vdash t : A$).

For readability, when $\Gamma_1$ and $\Gamma_2$ are understood we will write $\Gamma_\mathrm{p}$ for $\mathsf{Pack}\ \Gamma_1\ \Gamma_2$.

Implicitly, whenever we use the notation $\mathsf{Pack}\ \Gamma_1\ \Gamma_2$ it means that the two contexts are of the same length and well-formed with the same sorts. We can now state the fundamental lemma.

**Lemma 3.4** (Fundamental lemma). *Let $t_1$ and $t_2$ be two terms. If $\Gamma, \Gamma_1 \vdash t_1 : T_1$ and $\Gamma, \Gamma_2 \vdash t_2 : T_2$ and $t_1 \sim t_2$ then there exists $p$ such that $\Gamma, \mathsf{Pack}\ \Gamma_1\ \Gamma_2 \vdash p : t_1[\gamma_1]\ {}_{T_1[\gamma_1]}{\cong}_{T_2[\gamma_2]}\ t_2[\gamma_2]$.*

*Proof.* The proof is by induction on the derivation of $t_1 \sim t_2$. We show the three most interesting cases:

- Var

$$\frac{}{x \sim x}$$

$$\frac{t_1 \sqsubset t_2}{t_1 \sqsubset p_*\ t_2}$$

$$\frac{}{x \sqsubset x} \qquad \frac{A_1 \sqsubset A_2 \qquad B_1 \sqsubset B_2}{\Pi(x : A_1).\ B_1 \sqsubset \Pi(x : A_2).\ B_2}$$

$$\frac{A_1 \sqsubset A_2 \qquad B_1 \sqsubset B_2}{\Sigma(x : A_1).\, B_1 \sqsubset \Sigma(x : A_2).\, B_2}$$

$$\frac{A_1 \sqsubset A_2 \qquad u_1 \sqsubset u_2 \qquad v_1 \sqsubset v_2}{u_1 =_{A_1} v_1 \sqsubset u_2 =_{A_2} v_2} \qquad \frac{}{s \sqsubset s}$$

$$\frac{A_1 \sqsubset A_2 \qquad B_1 \sqsubset B_2 \qquad t_1 \sqsubset t_2}{\lambda(x : A_1).B_1.t_1 \sqsubset \lambda(x : A_2).B_2.t_2}$$

$$\frac{t_1 \sqsubset t_2 \qquad A_1 \sqsubset A_2 \qquad B_1 \sqsubset B_2 \qquad u_1 \sqsubset u_2}{t_1\ @_{x:A_1.B_1}\ u_1 \sqsubset t_2\ @_{x:A_2.B_2}\ u_2}$$

$$\frac{A_1 \sqsubset A_2 \qquad B_1 \sqsubset B_2 \qquad t_1 \sqsubset t_2 \qquad u_1 \sqsubset u_2}{\langle t_1; u_1\rangle_{x:A_1.B_1} \sqsubset \langle t_2; u_2\rangle_{x:A_2.B_2}}$$

$$\frac{A_1 \sqsubset A_2 \qquad B_1 \sqsubset B_2 \qquad p_1 \sqsubset p_2}{\pi_1^{x:A_1.B_1}\ p_1 \sqsubset \pi_1^{x:A_2.B_1}\ p_2}$$

$$\frac{A_1 \sqsubset A_2 \qquad B_1 \sqsubset B_2 \qquad p_1 \sqsubset p_2}{\pi_2^{x:A_1.B_1}\ p_1 \sqsubset \pi_2^{x:A_2.B_2}\ p_2} \qquad \frac{A_1 \sqsubset A_2 \qquad u_1 \sqsubset u_2}{\mathsf{refl}_{A_1}\ u_1 \sqsubset \mathsf{refl}_{A_2}\ u_2}$$

$$\frac{A_1 \sqsubset A_2 \qquad B_1 \sqsubset B_2 \qquad f_1 \sqsubset f_2 \qquad g_1 \sqsubset g_2 \qquad e_1 \sqsubset e_2}{\mathsf{funext}(x : A_1, B_1, f_1, g_1, e_1) \sqsubset \mathsf{funext}(x : A_2, B_2, f_2, g_2, e_2)}$$

$$\frac{A_1 \sqsubset A_2 \qquad u_1 \sqsubset u_2 \qquad v_1 \sqsubset v_2 \qquad p_1 \sqsubset p_2 \qquad q_1 \sqsubset q_2}{\mathsf{uip}(A_1, u_1, v_1, p_1, q_1) \sqsubset \mathsf{uip}(A_2, u_2, v_2, p_2, q_2)}$$

$$\frac{\begin{array}{c} A_1 \sqsubset A_2 \\ u_1 \sqsubset u_2 \qquad P_1 \sqsubset P_2 \qquad w_1 \sqsubset w_2 \qquad v_1 \sqsubset v_2 \qquad p_1 \sqsubset p_2 \end{array}}{\mathsf{J}(A_1, u_1, x.e.P_1, w_1, v_1, p_1) \sqsubset \mathsf{J}(A_2, u_2, x.e.P_2, w_2, v_2, p_2)}$$

**Figure 5.** Relation $\sqsubset$

If $x$ belongs to $\Gamma$, we apply reflexivity—together with uniqueness of typing (2.2)—to conclude. Otherwise, $\mathsf{Proj_e}\ x$ has the expected type (since $x[\gamma_1] \equiv \mathsf{Proj_1}\ x$ and $x[\gamma_2] \equiv \mathsf{Proj_2}\ x$).

- Application

$$\frac{t_1 \sim t_2 \qquad A_1 \sim A_2 \qquad B_1 \sim B_2 \qquad u_1 \sim u_2}{t_1\ @_{x:A_1.B_1}\ u_1 \sim t_2\ @_{x:A_2.B_2}\ u_2}$$

We have $\Gamma, \Gamma_1 \vdash t_1 @_{x:A_1.B_1} u_1 : T_1$ and $\Gamma, \Gamma_2 \vdash t_2 @_{x:A_2.B_2} u_2 : T_2$ which means by inversion (2.3) that the subterms are well-typed. We apply the induction hypothesis and then conclude.

- TransportLeft

$$\frac{t_1 \sim t_2}{p_*\ t_1 \sim t_2}$$

We have $\Gamma, \Gamma_1 \vdash p_*\ t_1 : T_1$ and $\Gamma, \Gamma_2 \vdash t_2 : T_2$. By inversion (2.3) we have $\Gamma, \Gamma_1 \vdash p : T_1' = T_1$ and $\Gamma, \Gamma_1 \vdash t_1 : T_1'$. By induction hypothesis we have $e$ such that $\Gamma, \Gamma_\mathsf{p} \vdash e : t_1[\gamma_1] \cong t_2[\gamma_2]$. From transitivity and symmetry we only need to provide a proof of $t_1[\gamma_1] \cong p[\gamma_1]_*\ t_1[\gamma_1]$ which is inhabited by $\langle p[\gamma_1]; \mathsf{refl}\ (p[\gamma_1]_*\ t_1[\gamma_1])\rangle_{\_.\_}$.

□

We can also prove that $\sim$ preserves substitution.

**Lemma 3.5.** *If $t_1 \sim t_2$ and $u_1 \sim u_2$ then $t_1[x \leftarrow u_1] \sim t_2[x \leftarrow u_2]$.*

*Proof.* We proceed by induction on the derivation of $t_1 \sim t_2$. □

## 4 Translating ETT to ITT

### 4.1 The Translation

We now define the translations (let us stress the plural here) of an extensional judgment. We extend $\sqsubset$ canonically to contexts ($\Gamma \sqsubset \overline{\Gamma}$ when they bind the same variables and the types are in relation for $\sqsubset$).

Before defining the translation, we define a set $[\![\Gamma \vdash_\mathsf{x} t : A]\!]$ of typing judgments in ITT associated to a typing judgment $\Gamma \vdash_\mathsf{x} t : A$ in ETT. The idea is that this set describes all the possible translations that lead to the expected property. When $\overline{\Gamma} \vdash \overline{t} : \overline{A} \in [\![\Gamma \vdash_\mathsf{x} t : A]\!]$, we say that $\overline{\Gamma} \vdash \overline{t} : \overline{A}$ realises $\Gamma \vdash_\mathsf{x} t : A$. The translation will be given by showing that this set is inhabited by induction on the derivation.

**Definition 4.1** (Characterisation of possible translations)**.**

- For any $\vdash_\mathsf{x} \Gamma$ we define $[\![\vdash_\mathsf{x} \Gamma]\!]$ as a set of valid judgments (in ITT) such that $\vdash \overline{\Gamma} \in [\![\vdash_\mathsf{x} \Gamma]\!]$ if and only if $\Gamma \sqsubset \overline{\Gamma}$.
- Similarly, $\overline{\Gamma} \vdash \overline{t} : \overline{A} \in [\![\Gamma \vdash_\mathsf{x} t : A]\!]$ iff $\vdash \overline{\Gamma} \in [\![\vdash_\mathsf{x} \Gamma]\!]$ and $A \sqsubset \overline{A}$ and $t \sqsubset \overline{t}$.

In order to better master the shape of the produced realiser, we state the following lemma which shows that it has the same head type constructor as the type it realises. This is important for instance for the case of an application, where we do not know a priori if the translated function has a dependent product type, which is required to be able to use the typing rule for application.

**Lemma 4.2.** *We can always* choose *types $\overline{T}$ that have the same head constructor as $T$.*

*Proof.* Assume we have $\overline{\Gamma} \vdash \overline{t} : \overline{T} \in [\![\Gamma \vdash_x t : T]\!]$. By definition of $\sqsubset$, $T \sqsubset \overline{T}$ means that $\overline{T}$ is shaped $p_*\ q_*\ \ldots\ r_*\ \overline{T}'$ with $\overline{T}'$ having the same head constructor as $T$. By inversion (2.3), the subterms are typable, including $\overline{T}'$. Actually, from inversion, we even get that the type of $\overline{T}'$ is a universe. Then, using lemma 3.4 and lemma 3.3, we get $\overline{\Gamma} \vdash e : \overline{T} = \overline{T}'$. We conclude with $\overline{\Gamma} \vdash e_*\ \overline{t} : \overline{T}' \in [\![\Gamma \vdash_x t : T]\!]$. □

Finally, in order for the induction to go through, we need to know that when we have a realiser of a derivation $\Gamma \vdash_x t : T$, we can pick an arbitrary other type realising $\Gamma \vdash_x T$ and still get a new derivation realising $\Gamma \vdash_x t : T$ with that type. This is important for instance for the case of an application, where the type of the domain of the translated function may differ from the type of the translated argument. So we need to be able to change it *a posteriori*.

**Lemma 4.3.** *When we have $\overline{\Gamma} \vdash \overline{t} : \overline{T} \in [\![\Gamma \vdash_x t : T]\!]$ and $\overline{\Gamma} \vdash \overline{T}' \in [\![\Gamma \vdash_x T]\!]$ then we also have $\overline{\Gamma} \vdash \overline{t}' : \overline{T}' \in [\![\Gamma \vdash_x t : T]\!]$ for some $\overline{t}'$.*

*Proof.* By definition we have $T \sqsubset \overline{T}$ and $T \sqsubset \overline{T}'$ and thus $T \sim \overline{T}$ and $T \sim \overline{T}'$, implying $\overline{T} \sim \overline{T}'$ by transitivity (3.2). By lemma 3.4 (in the case $\Gamma_1 \equiv \Gamma_2 \equiv \bullet$) we get $\overline{\Gamma} \vdash p : \overline{T} \cong \overline{T}'$ for some $p$. By lemma 3.3 (and lemma 4.2 to give universes as types to $\overline{T}$ and $\overline{T}'$) we can assume $\overline{\Gamma} \vdash p : \overline{T} = \overline{T}'$. Then $\overline{\Gamma} \vdash p_*\ \overline{t} : \overline{T}'$ is still a translation since $\sqsubset$ ignores transports. □

We can now define the translation. This is done by mutual induction on context well-formedness, typing and conversion derivations. Indeed, in order to be able to produce a realiser by induction, we need to show that every conversion in ETT is translated as an heterogeneous equality in ITT.

**Theorem 4.4** (Translation)**.**

- *If $\vdash_x \Gamma$ then there exists $\vdash \overline{\Gamma} \in [\![\vdash_x \Gamma]\!]$,*
- *If $\Gamma \vdash_x t : T$ then for any $\vdash \overline{\Gamma} \in [\![\vdash_x \Gamma]\!]$ there exist $\overline{t}$ and $\overline{T}$ such that $\overline{\Gamma} \vdash \overline{t} : \overline{T} \in [\![\Gamma \vdash_x t : T]\!]$,*
- *If $\Gamma \vdash_x u \equiv v : A$ then for any $\vdash \overline{\Gamma} \in [\![\vdash_x \Gamma]\!]$ there exist $A \sqsubset \overline{A}, A \sqsubset \overline{A}', u \sqsubset \overline{u}, v \sqsubset \overline{v}$ and $\overline{e}$ such that $\overline{\Gamma} \vdash \overline{e} : \overline{u}\ {}_{\overline{A}}{\cong}_{\overline{A}'}\ \overline{v}$.*

*Proof.* We prove the theorem by induction on the derivation in the extensional type theory. We only show the two most interesting cases of application and conversion.

- Application
$$\frac{\Gamma \vdash_x A : s \qquad \Gamma, x : A \vdash_x B : s' \qquad \Gamma \vdash_x t : \Pi(x:A).\ B \qquad \Gamma \vdash_x u : A}{\Gamma \vdash_x t\ @_{x:A.B}\ u : B[x \leftarrow u]}$$
Using IH together with lemmata 4.2 and 4.3 we get $\overline{\Gamma} \vdash \overline{A} : s$ and $\overline{\Gamma}, x : \overline{A} \vdash \overline{B} : s'$ and $\overline{\Gamma} \vdash \overline{t} : \Pi(x : \overline{A}).\ \overline{B}$ and $\overline{\Gamma} \vdash \overline{u} : \overline{A}$ meaning we can conclude $\overline{\Gamma} \vdash \overline{t} @_{x:\overline{A}.\overline{B}} \overline{u} : \overline{B}[x \leftarrow \overline{u}] \in [\![\Gamma \vdash_x t\ @_{x:A.B}\ u : B[x \leftarrow u]]\!]$.
- Conversion
$$\frac{\Gamma \vdash_x u : A \qquad \Gamma \vdash_x A \equiv B}{\Gamma \vdash_x u : B}$$
By IH and lemma 3.3 we have $\overline{\Gamma} \vdash \overline{e} : \overline{A} = \overline{B}$ which implies $\overline{\Gamma} \vdash \overline{A} \in [\![\Gamma \vdash_x A]\!]$ by inversion (2.3), thus, from lemma 4.3 and IH we get $\overline{\Gamma} \vdash \overline{u} : \overline{A}$, yielding $\overline{\Gamma} \vdash \overline{e}_*\ \overline{u} : \overline{B} \in [\![\Gamma \vdash_x u : B]\!]$.

□

## 4.2 Meta-theoretical Consequences

We can check that all ETT theorems whose type are typable in ITT have proofs in ITT as well:

**Corollary 4.5** (Preservation of ITT)**.** *If $\vdash_x t : T$ and $\vdash T$ then there exist $\overline{t}$ such that $\vdash \overline{t} : T \in [\![\vdash_x t : T]\!]$.*

*Proof.* Since $\vdash \bullet \in [\![\vdash_x \bullet]\!]$, by Theorem (4.4), there exists $\overline{t}$ and $\overline{T}$ such that $\vdash \overline{t} : \overline{T} \in [\![\vdash_x t : T]\!]$ But as $\vdash T$, we have $\vdash T \in [\![\vdash_x T]\!]$, and, using Lemma 4.3, we obtain $\vdash \overline{t} : T \in [\![\vdash_x t : T]\!]$. □

**Corollary 4.6** (Relative consistency)**.** *Assuming ITT is consistent, there is no term $t$ such that $\vdash_x t : \Pi(A : \square_0).\ A$.*

*Proof.* Assume such a $t$ exists. By the Corollary 4.5, because $\vdash \Pi(A : \square_0).\ A$, there exists $\overline{t}$ such that $\vdash \overline{t} : \Pi(A : \square_0).\ A$ which contradicts the assumed consistency of ITT. □

## 4.3 Optimisations

Up until now, we remained silent about one thing: the size of the translated terms. Indeed, the translated term is a decoration of the initial one by transports which appear in many locations. For example, at each application we use a transport by lemma 4.2 to ensure that the term in function position is given a function type. In most cases—in particular when translating ITT terms—this produces unnecessary transports (often by reflexivity) that we wish to avoid.

In order to limit the size explosion, in the above we use a different version of transport, namely transport′ such that

$$\begin{aligned} \mathsf{transport}'_{A_1,A_2}(p,t) &= t && \text{when } A_1 =_\alpha A_2 \\ &= p_* t && \text{otherwise.} \end{aligned}$$

The idea is that we avoid *trivially* unnecessary transports (we do not deal with $\beta$-conversion for instance). We extend this technique to the different constructors of equality (symmetry, transitivity, …) so that they reduce to reflexivity whenever possible. Take transitivity for instance:

$$\begin{aligned} \mathsf{transitivity}'(\mathsf{refl}\ u, q) &= q \\ \mathsf{transitivity}'(p, \mathsf{refl}\ u) &= p \\ \mathsf{transitivity}'(p,q) &= \mathsf{transitivity}(p,q). \end{aligned}$$

We show these *defined terms* enjoy the same typing rules as their counterparts and use them instead. In practice it is enough to recover the exact same term when it is typed in ITT.

## 5 Formalisation with Template-Coq

We have formalised the translation in the setting of TemplateCoq [Anand et al. 2018] in order to have a more precise proof, but also to evidence the fact that the translation is indeed constructive and can be used to perform computations.

TemplateCoq is a Coq library that has a representation of Coq terms as they are in Coq's kernel (in particular using de Bruijn indices for variables) and a (partial) implementation of the type checking algorithm (not checking guardedness of fixpoints or positivity of inductive types). It comes with a Coq plugin that permits to quote Coq terms into their representations, and to produce Coq terms from their representation (if they indeed denote well-typed terms). We have integrated our formalisation within that framework in order to ensure our presentations of ETT and ITT are close to Coq, but also to take advantage of the quoting mechanism to produce terms using the interactive mode (in particular we get to use tactics). Note that we also rely on Mangin and Sozeau's Equations [Sozeau 2010] plugin to derive nice dependent induction principles.

Our formalisation takes full advantage of its easy interfacing with TemplateCoq: we define two theories, namely ETT and ITT, but ITT enjoys a lot of syntactic sugar by having things such as transport, heterogeneous equality and packing as part of the syntax. The operations regarding these constructors—in particular the tedious ones—are written in Coq and then quoted to finally be *realised* in the translation from ITT to TemplateCoq.

***Interoperability with TemplateCoq.*** The translation we define from ITT to TemplateCoq is not proven correct, but it is not really important as it can just be seen as a feature to observe the produced terms in a nicer setting. In any case, TemplateCoq does not yet provide a complete formalisation of CIC rules, as guard checking of recursive definitions and strict positivity of inductive type declarations are not formalised yet.

Our formalised theorems however do not depend on TemplateCoq itself and as such there is no need to *trust* the plugin.

We also provide a translation from TemplateCoq to ETT that we will describe more extensively with the examples (Section 5.4).

### 5.1 Quick Overview of the Formalisation

The file SAst.v contains the definition of the (common) abstract syntax of ETT and ITT in the form of an inductive definition with de Bruijn indices for variables (like in TemplateCoq). Sorts are defined separately in Sorts.v and we will address them later in Section 5.3.

```
Inductive sterm : Type :=
| sRel (n : nat)
| sSort (s : sort)
| sProd (nx : name) (A B : sterm)
| sLambda (nx : name) (A B t : sterm)
| sApp (u : sterm) (nx : name) (A B v : sterm)
| sEq (A u v : sterm)
| sRefl (A u : sterm)
| (* ... *) .
```

The files ITyping.v and XTyping.v define respectively the typing judgments for ITT and ETT, using mutual inductive types. Then, most of the files are focused on the meta-theory of ITT and can be ignored by readers who don't need to see yet another proof of subject reduction.

The most interesting files are obviously those where the fundamental lemma and the translation are formalised: FundamentalLemma.v and Translation.v. For instance, here is the main theorem, as stated in our formalisation:

```
Theorem complete_translation Σ :
  type_glob Σ ->
  (forall Γ (h : XTyping.wf Σ Γ), ∑ Γ', Σ |--i Γ' # ⟦ Γ ⟧ ) *
  (forall Γ t A (h : Σ ;;; Γ |-x t : A)
   Γ' (hΓ : Σ |--i Γ' # ⟦ Γ ⟧),
    ∑ A' t', Σ ;;;; Γ' |--- [t'] : A' # ⟦ Γ |--- [t] : A ⟧) *
  (forall Γ u v A (h : Σ ;;; Γ |-x u = v : A)
   Γ' (hΓ : Σ |--i Γ' # ⟦ Γ ⟧),
    ∑ A' A'' u' v' p', eqtrans Σ Γ A u v Γ' A' A'' u' v' p').
```

Herein `type_glob Σ` refers to the fact that some global context is well-typed, its purpose is detailed in Section 5.2. The fact that the theorem holds in Coq ensures we can actually compute a translated term and type out of a derivation in ETT.

### 5.2 Inductive Types and Recursion

In the proof of Section 4, we didn't mention anything about inductive types, pattern-matching or recursion as it is a bit technical on paper. In the formalisation, we offer a way to still be able to use them, and we will even show how it works in practice with the examples (Section5.4).

The main guiding principle is that inductive types and induction are orthogonal to the translation, they should more or less be translated to themselves. To realise that easily, we just treat an inductive definition as a way to introduce new constants in the theory, one for the type, one for each constructor, one for its elimination principle, and one equality per computation rule. For instance, the natural numbers can be represented by having the following constants in the context:

$$
\begin{aligned}
&\mathsf{nat} && : \square_0 \\
&0 && : \mathsf{nat} \\
&\mathsf{S} && : \mathsf{nat} \to \mathsf{nat} \\
&\mathsf{natrec} && : \forall P,\ P\ 0 \to (\forall m,\ P\ m \to P\ (\mathsf{S}\ m)) \to \forall n,\ P\ n \\
&\mathsf{natrec}_0 && : \forall P\ P_z\ P_s,\ \mathsf{natrec}\ P\ P_z\ P_s\ 0 = P_z \\
&\mathsf{natrec}_\mathsf{S} && : \forall P\ P_z\ P_s\ n, \\
& && \quad \mathsf{natrec}\ P\ P_z\ P_s\ (\mathsf{S}\ n) = P_s\ n\ (\mathsf{natrec}\ P\ P_z\ P_s\ n)
\end{aligned}
$$

Here we rely on the reflection rule to obtain the computational behaviour of the eliminator natrec.

This means for instance that we do not consider inductive types that would only make sense in ETT, but we deem this not to be a restriction and to the best of our knowledge isn't something that is usually considered in the literature. With that in mind, our translation features a global context of typed constants with the restriction that the types of those constants should be well-formed in ITT. Those constants are thus used as black boxes inside ETT.

With this we are able to recover what we were missing from Coq, without having to deal with the trouble of proving that the translation doesn't break the guard condition of fixed points, and we are instead relying on a more type-based approach.

## 5.3 About Universes and Homotopy

The experienced reader might have noticed that our treatment of universes (except perhaps for the absence of cumulativity) was really superficial and the notion of sorts used is rather orthogonal to our main development. This is even more apparent in the formalisation. Indeed, we didn't fix a specific universe hierarchy, but instead specify what properties it should have, in what is reminiscent to a (functional[5]) PTS formulation.

```
Class Sorts.notion := {
  sort : Type ;
  succ : sort -> sort ;
  prod_sort : sort -> sort -> sort ;
  sum_sort : sort -> sort -> sort ;
  eq_sort : sort -> sort ;
  eq_dec : forall s z : sort, {s = z} + {s <> z} ;
  succ_inj : forall s z, succ s = succ z -> s = z
}.
```

From the notion of sorts, we require functions to get the sort of a sort, the sort of a product from the sorts of its arguments, and (crucially) the sort of an identity type. We also require some measure of decidable equality and injectivity on those.

This allows us to instantiate this by a lot of different notions including the one presented earlier in the paper or even its extension with a universe Prop of propositions (like CIC [Bertot and Castéran 2004]). We present here two instances that have their own interest.

**Type *in* Type.** One of the instances we provide is one with only one universe Type, with the inconsistent typing rule Type : Type. Although inconsistent, this allows us to interface with TemplateCoq, without the—for the time being—very time-consuming universe constraint checking.

***Homotopy Type System and Two-Level Type Theory.*** Another interesting application (or rather instance) of our formalisation is a translation from Homotopy Type System (HTS) [Voevodsky 2013] to Two-Level Type Theory (2TT) [Altenkirch et al. 2016; Annenkov et al. 2017].

HTS and 2TT arise from the incompatibility between UIP—recall it is provable in ETT—and univalence. The idea is to have two distinct notions of equality in the theory, a *strict* one satisfying UIP, and a *fibrant* one corresponding to the homotopy type theory equality, possibly satisfying univalence. This actually induces a separation in the types of the theory: some of them are called *fibrant* and the fibrant or homotopic equality can only be eliminated on those. HTS can be seen as an extension of 2TT with reflection on the strict equality just like ETT is an extension of ITT.

We can recover HTS and 2TT in our setting by taking $\mathsf{F}_i$ and $\mathsf{U}_i$ as respectively the fibrant and strict universes of those theories (for $i \in \mathbb{N}$), along with the following PTS rules:

$$\begin{array}{llll}
(\mathsf{F}_i, \mathsf{F}_{i+1}) & \in \mathsf{Ax} & (\mathsf{U}_i, \mathsf{U}_{i+1}) & \in \mathsf{Ax} \\
(\mathsf{F}_i, \mathsf{F}_j, \mathsf{F}_{\max(i,j)}) & \in \mathsf{R} & (\mathsf{F}_i, \mathsf{U}_j, \mathsf{U}_{\max(i,j)}) & \in \mathsf{R} \\
(\mathsf{U}_i, \mathsf{F}_j, \mathsf{U}_{\max(i,j)}) & \in \mathsf{R} & (\mathsf{U}_i, \mathsf{U}_j, \mathsf{U}_{\max(i,j)}) & \in \mathsf{R}
\end{array}$$

and the fact that the sort of the (strict) identity type on $A : s$ is the *strictified* version of $s$, *i.e.,* $\mathsf{U}_i$ for $s = \mathsf{U}_i$ or $s = \mathsf{F}_i$. In order to have the fibrant equality, one simply needs to do as in Section 5.2.

In short, the translation from HTS to 2TT is basically the same as the one from ETT to ITT we presented in this paper, and this fact is factorised through our formalisation.

## 5.4 ETT-flavoured Coq: Examples

In this section we demonstrate how our translation can bring extensionality to the world of Coq in action. The examples can be found in plugin_demo.v.

***First, a pedestrian approach.*** We would like to begin by showing how one can write an example step by step before we show how it can be instrumented and automated as a plugin. For this we use a self-contained example without any inductive types or recursion, illustrating a very simple case of reflection. The term we want to translate is our introductory example of transport:

$$\lambda\, A\, B\, e\, x.\, x : \Pi\, A\, B.\, A = B \to A \to B$$

which relies on the equality $e : A = B$ and reflection to convert $x : A$ to $x : B$. Of course, this definition isn't accepted in Coq because this conversion is not valid in ITT.

```
Fail Definition pseudoid (A B : Type) (e : A = B) (x : A) : B := x.
```

However, we still want to be able to write it *in some way*, in order to avoid manipulating de Bruijn indices directly. For this, we use a little trick by first defining a Coq axiom to represent an ill-typed term:

```
Axiom candidate : forall A B (t : A), B.
```

`candidate A B t` is a candidate `t` of type `A` to inhabit type `B`. We complete this by adding a notation that is reminiscent to Agda's [Norell 2007] hole mechanism.

```
Notation "'{!' t '!}'" := (candidate _ _ t).
```

We can now write the ETT function within Coq.

```
Definition pseudoid (A B : Type) (e : A = B) (x : A) : B := {! x !}.
```

[5]Meaning the sort of a sort, and the sort of a product are functions, necessary to the uniqueness of types (2.2).

We can then quote the term and its type to TemplateCoq thanks to the `Quote Definition` command provided by the plugin.

```
Quote Definition pseudoid_term :=
  ltac:(let t := eval compute in pseudoid in exact t).
Quote Definition pseudoid_type :=
  ltac:(let T := type of pseudoid in exact T).
```

The terms that we get are now TemplateCoq terms, representing Coq syntax. We need to put them in ETT, meaning adding the annotations, and also removing the `candidate` axiom. This is the purpose of the `fullquote` function that we provide in our formalisation.

```
Definition pretm_pseudoid :=
  Eval lazy in fullquote (2^18) Σ [] pseudoid_term empty empty nomap.
Definition tm_pseudoid :=
  Eval lazy in match pretm_pseudoid with
              | Success t => t
              | Error _ => sRel 0
              end.

Definition prety_pseudoid :=
  Eval lazy in fullquote (2^18) Σ [] pseudoid_type empty empty nomap.
Definition ty_pseudoid :=
  Eval lazy in match prety_pseudoid with
               | Success t => t
               | Error _ => sRel 0
               end.
```

`tm_pseudoid` and `ty_pseudoid` correspond respectively to the ETT representation of `pseudoid` and its type. We then produce, using our home-brewed Ltac type-checking tactic, the corresponding ETT typing derivation (notice the use of reflection to typecheck).

```
Lemma type_pseudoid : Σi ;;; [] |-x tm_pseudoid : ty_pseudoid.
Proof.
  unfold tm_pseudoid, ty_pseudoid.
  ettcheck. cbn.
  eapply reflection with (e := sRel 1).
  ettcheck.
Defined.
```

We can then translate this derivation, obtain the translated term and then convert it to TemplateCoq.

```
Definition itt_pseudoid : sterm :=
  Eval lazy in
  let '(_ ; t ; _) :=
    type_translation type_pseudoid istrans_nil
  in t.

Definition tc_pseudoid : tsl_result term :=
  Eval lazy in
  tsl_rec (2 ^ 18) Σ [] itt_pseudoid empty.
```

Once we have it, we *unquote* the term to obtain a Coq term (notice that the only use of reflection has been replaced by a transport).

```
fun (A B : Type) (e : A = B) (x : A) => transport e x
     : forall A B : Type, A = B -> A -> B
```

***Making a Plugin with TemplateCoq.*** All of this work is pretty systematic. Fortunately for us, TemplateCoq also features a monad to reify Coq commands which we can use to *program* the translation steps. As such we have written a complete procedure, relying on type checkers we wrote for ITT and ETT, which can generate equality obligations.

Thanks to this, the user doesn't have to know about the details of implementation of the translation, and stay within the Coq ecosystem.

For instance, our previous example now becomes:

```
Definition pseudoid (A B : Type) (e : A = B) (x : A) : B := {! x !}.

Run TemplateProgram (Translate ε "pseudoid").
```

This produces a Coq term `pseudoid'` corresponding to the translation (ε is the empty translation context, see the next example to understand the need for a translation context). Notice how the user doesn't even have to provide any proof of equality or derivations of any sort. The derivation part is handled by our own typechecker while the obligation part is solved automatically by the Coq obligation mechanism.

***About inductive types.*** As we promised, our translation is able to handle inductive types. For this consider the inductive type of vectors (or length-indexed lists) below, together with a simple definition (we will remain in ITT for simplicity).

```
Inductive vec A : nat -> Type :=
| vnil : vec A 0
| vcons : A -> forall n, vec A n -> vec A (S n).

Arguments vnil {_}.
Arguments vcons {_} _ _ _.

Definition vv := vcons 1 _ vnil.
```

This time, in order to apply the translation we need to extend the translation context with `nat` and `vec`.

```
Run TemplateProgram (
    Θ <- TranslateConstant ε "nat" ;;
    Θ <- TranslateConstant Θ "vec" ;;
    Translate Θ "vv"
).
```

The command `TranslateConstant` enriches the current translation context with the types of the inductive type and of its constructors. The translation context then also contains associative tables between our own representation of constants and those of Coq. Unsurprisingly, the translated Coq term is the same as the original term.

***Reversal of vectors.*** Next, we tackle a motivating example: reversal on vectors. Indeed, implementing this operation the same way it can be done on lists ends up in the following conversion problem:

```
Fail Definition vrev {A n m} (v : vec A n) (acc : vec A m)
: vec A (n + m) :=
  vec_rect A (fun n _ => forall m, vec A m -> vec A (n + m))
           (fun m acc => acc)
           (fun a n _ rv m acc => rv _ (vcons a m acc))
           n v m acc.
```

The recursive call returns a vector of length `n + S m` where the context expects one of length `S n + m`. In ITT, these types are not convertible. This example is thus a perfect fit for ETT where we can use the fact that these two expressions always compute to the same thing when instantiated with concrete numbers.

```
Definition vrev {A n m} (v : vec A n) (acc : vec A m)
: vec A (n + m) :=
  vec_rect A (fun n _ => forall m, vec A m -> vec A (n + m))
           (fun m acc => acc)
           (fun a n _ rv m acc => {! rv _ (vcons a m acc) !})
           n v m acc.

Run TemplateProgram (
    Θ <- TranslateConstant ε "nat" ;;
    Θ <- TranslateConstant Θ "vec" ;;
    Θ <- TranslateConstant Θ "Nat.add" ;;
    Θ <- TranslateConstant Θ "vec_rect" ;;
    Translate Θ "vrev"
).
```

This generates four obligations that are all solved automatically. One of them contains a proof of `S n + m = n + S m` while the remaining three correspond to the computation rules of addition (as mentioned before, add is simply a constant and does not compute in our representation, hence the need for equalities). The returned term is the following, with only one transport remaining (remember our interpretation map removes unnecessary transports).

```
fun (A : Type) (n m : nat) (v : vec A n) (acc : vec A m) =>
vec_rect A
  (fun n _ => forall m, vec A m -> vec A (n + m))
  (fun m acc => acc)
  (fun a n₀ v₀ rv m₀ acc₀ =>
    transport (vrev_obligation_3 A n m v acc a n₀ v₀ rv m₀ acc₀)
      (rv (S m₀) (vcons a m₀ acc₀))) n v m acc
: forall A n m, vec A n -> vec A m -> vec A (n + m)
```

### 5.5 Towards an Interfacing between Andromeda and Coq

Andromeda [Bauer et al. 2016] is a proof assistant implementing ETT in a sense that is really close to our formalisation. Aside from a concise nucleus with a basic type theory, most things happen with the declaration of constants with given types, including equalities to define the computational behaviour of eliminators for instance. This is essentially what we do in our formalisation. Furthermore, their theory relies on Type : Type, meaning, our modular handling of universes can accommodate for this as well.

All in all, it should be possible in the near future to use our translation to produce Coq terms out of Andromeda developments. Note that this would not suffer from the difficulties in generating typing derivations since Andromeda generates them.

### 5.6 Composition with other Translations

This translation also enables the formalisation of translations that target ETT rather than ITT and still get mechanised proofs of (relative) consistency by composition with this ETT to ITT translation. This could also be used to implement plugins based on the composition of translations. In particular, supposing we have a theory which forms a subset of ETT and whose conversion is decidable. Using this translation, we could formalise it as an embedded domain-specific type theory and provide an automatic translation of well-typed terms into witnesses in Coq. This would make it possible to extend conversion with the theory of lists for example.

This would provide a simple way to justify the consistency of CoqMT [Jouannaud and Strub 2017] for example, seeing it as an extensional type theory where reflection is restricted to equalities on a specific domain whose theory is decidable.

## 6 Limitations and Axioms

Currently, the representation of terms and derivations and the computational content of the proof only allow us to deal with the translation of relatively small terms but we hope to improve that in the future. As we have seen, the actual translation involves the computational content of lemmata of inversion, substitution, weakening and equational reasoning and thus cannot be presented as a simple recursive definition on derivations.

As we already mentioned, the axioms K and FunExt are both necessary in ITT if we want the translation to be conservative as they are provable in ETT [Hofmann 1995]. However, one might still be concerned about having axioms as they can for instance hinder canonicity of the system. In that respect, K isn't really a restriction since it preserves canonicity. The best proof of that is probably Agda itself which natively features K—in fact, one needs to explicitly deactivate it with a flag if they wish to work without.

The case of FunExt is trickier. It should be possible to realise the axiom by composing our translation with a setoid interpretation [Altenkirch 1999] which validates it, or by going into a system featuring it, for instance by implementing Observational Type Theory [Altenkirch et al. 2007] like EPIGRAM [McBride 2004].

However, these two axioms are not used to define the translation itself, but only to witness UIP and function extensionality in the translation to Coq. The translation only relies on one axiom, called `conv_trans_AXIOM` in the formalisation, stating that conversion of ITT is transitive. The proof of this property basically sums up to the confluence of the reduction rules of ITT which is out of scope for this paper and has recently been formalised in Agda [Abel et al. 2017] (in a simpler setting with only one universe). Regardless, this axiom inhabits a proposition (the type of conversion is in **`Prop`**) and is thus irrelevant for computation. Actually no information about the derivation leaks to the production of the ITT term.

On a different note, the `candidate` axiom allows us to derive `False` but is merely used to write ill-typed terms in Coq. The translated term will never make us of it and one can always check if a term is relying on unsafe assumptions thanks to the **`Print`** `Assumptions` command.

## 7 Related Works and Conclusion

The seminal works on the precise connection between ETT and ITT go back to Streicher [1993] and Hofmann [1995, 1997]. In particular, the work of Hofmann provides a categorical answer to the question of consistency and conservativity of ETT over ITT with UIP and FunExt. Ten years later, Oury [2005, 2006] provided a translation from ETT to ITT with UIP and FunExt and other axioms (mainly due to technical difficulties). Although a first step towards a move from categorical semantics to a syntactic translation, his work does not stress any constructive aspect of the proof and shows that there merely exist translations in ITT to a typed term in ETT.

van Doorn et al. [2013] have later proposed and formalised a similar translation between a PTS with and without explicit conversion. This does not entail anything about ETT to ITT but we can find similarities in that there is a witness of conversion between any term and itself under an explicit conversion, which internalises irrelevance of explicit conversions. This morally corresponds to a Uniqueness of Conversions principle.

The Program [Sozeau 2007] extension of Coq performs a related coercion insertion algorithm, between objects in subsets on the same carrier or in different instances of the same inductive family, assuming a proof-irrelevance axiom. Inserting coercions locally is not as general as the present translation from ETT to ITT which can insert transports in any context.

In this paper we provide the first effective translation from ETT to ITT with UIP and FunExt. The translation has been formalised in Coq using TemplateCoq, a meta-programming plugin of Coq. This translation is also effective in the sense that we can produce in the end a Coq term using the TemplateCoq denotation machinery. With ongoing work to extend the translation to the inductive fragment of Coq, we are paving the way to an extensional version of the Coq proof assistant which could be translated back to its intensional version, allowing the user to navigate between the two modes, and in the end produce a proof term checkable in the intensional fragment.

## Acknowledgments

We would like to thank Andrej Bauer and Philipp Haselwarter with whom we had fruitful discussions on the subject, prior to this work. We also would like to thank the attendees of the Aarhus EUTypes 2018 meeting for their insightful feedback on the plugin stemming from the translation.

## References

Andreas Abel, Joakim Öhman, and Andrea Vezzosi. 2017. Decidability of Conversion for Type Theory in Type Theory. *Proc. ACM Program. Lang.* 2, POPL, Article 23 (Dec. 2017), 29 pages. https://doi.org/10.1145/3158111

Stuart F. Allen, Robert L. Constable, Richard Eaton, Christoph Kreitz, and Lori Lorigo. 2000. The Nuprl Open Logical Environment. In *Automated Deduction - CADE-17, Pittsburgh, PA, USA, June 17-20, 2000, Proceedings (LNCS)*, David A. McAllester (Ed.), Vol. 1831. Springer, 170–176.

T. Altenkirch. 1999. Extensional equality in intensional type theory. In *Proceedings of LICS*. 412–420.

Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus. 2016. Extending Homotopy Type Theory with Strict Equality. (2016). arXiv:1604.03799

Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. 2007. Observational equality, now!. In *Proceedings of the 2007 workshop on Programming languages meets program verification*. ACM, 57–68.

Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. 2018. Towards Certified Meta-Programming with Typed Template-Coq. In *ITP 2018, Oxford, UK, July 9-12, 2018, Proceedings (Lecture Notes in Computer Science)*, Jeremy Avigad and Assia Mahboubi (Eds.), Vol. 10895. Springer, 20–39.

Danil Annenkov, Paolo Capriotti, and Nicolai Kraus. 2017. Two-Level Type Theory and Applications. *CoRR* abs/1705.03307 (2017). arXiv:1705.03307

Andrej Bauer, Gaëtan Gilbert, Philipp G. Haselwarter, Matija Pretnar, and Chris Stone. 2016. The 'Andromeda' prover. http://www.andromeda-prover.org/

Jean-philippe Bernardy, Patrik Jansson, and Ross Paterson. 2012. Proofs for free: Parametricity for dependent types. *Journal of Functional Programming* 22, 2 (2012), 107–152.

Yves Bertot and Pierre Castéran. 2004. Interactive Theorem Proving and Program Development.

Marc Bezem, Thierry Coquand, and Simon Huber. 2013. A Model of Type Theory in Cubical Sets. (December 2013). http://www.cse.chalmers.se/~coquand/mod1.pdf

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

The Coq development team. 2017. *The Coq proof assistant reference manual*. LogiCal Project. http://coq.inria.fr Version 8.7.

Martin Hofmann. 1995. Conservativity of equality reflection over intensional type theory. In *International Workshop on Types for Proofs and Programs*. Springer, 153–164.

Martin Hofmann. 1997. *Extensional constructs in intensional type theory*. Springer.

Martin Hofmann and Thomas Streicher. 1998. The Groupoid Interpretation of Type Theory. In *Twenty-five years of constructive type theory (Venice, 1995)*. Vol. 36. Oxford Univ. Press, New York, 83–111.

Jean-Pierre Jouannaud and Pierre-Yves Strub. 2017. Coq without Type Casts: A Complete Proof of Coq Modulo Theory. In *LPAR-21, Maun, Botswana, May 7-12, 2017 (EPiC Series in Computing)*, Thomas Eiter and David Sands (Eds.), Vol. 46. EasyChair, 474–489.

Chris Kapulkin and Peter LeFanu Lumsdaine. 2012. The simplicial model of univalent foundations. *arXiv preprint arXiv:1211.2851* (2012).

Conor McBride. 2000. *Dependently typed functional programs and their proofs*. Ph.D. Dissertation. University of Edinburgh.

Conor McBride. 2004. Epigram: Practical programming with dependent types. In *International School on Advanced Functional Programming*. Springer, 130–170.

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Vol. 32. Citeseer.

Nicolas Oury. 2005. Extensionality in the calculus of constructions. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 278–293.

Nicolas Oury. 2006. *Egalité et filtrage avec types dépendants dans le calcul des constructions inductives*. Ph.D. Dissertation. http://www.theses.fr/2006PA112136

Matthieu Sozeau. 2007. Program-ing Finger Trees in Coq. In *ICFP'07*. ACM Press, Freiburg, Germany, 13–24.

Matthieu Sozeau. 2010. Equations: A Dependent Pattern-Matching Compiler. In *ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings (Lecture Notes in Computer Science)*, Matt Kaufmann and Lawrence C. Paulson (Eds.), Vol. 6172. Springer, 419–434.

Thomas Streicher. 1993. *Investigations into intensional type theory*.

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

Floris van Doorn, Herman Geuvers, and Freek Wiedijk. 2013. Explicit convertibility proofs in pure type systems. In *Proceedings of the Eighth ACM SIGPLAN international workshop on Logical frameworks & meta-languages: theory & practice*. ACM, 25–36.

Vladimir Voevodsky. 2013. A simple type system with two identity types. https://ncatlab.org/homotopytypetheory/files/HTS.pdf