Equivalences for Free!

Univalent Parametricity for Effective Transport

Nicolas Tabareau
Gallinette Project-Team, Inria
Nantes, France

Éric Tanter
Pleiad, DCC - University of Chile
Santiago, Chile

Matthieu Sozeau
Pi.R2 Project-Team, Inria and IRIF
Paris, France

Abstract
Homotopy Type Theory promises a unification of the concepts of equality and equivalence in Type Theory, through the introduction of the univalence principle. However, existing proof assistants based on type theory treat this principle as an axiom, and it is not yet entirely clear how to extend them to handle univalence internally. In this paper, we propose a construction grounded on a univalent version of parametricity to bring the benefits of univalence to the programmer and prover, that can be used on top of existing type theories. In particular, univalent parametricity strengthens parametricity to ensure preservation of type equivalences. We present a lightweight framework implemented in the Coq proof assistant that allows the user to transparently transfer definitions and theorems for a type to an equivalent one, as if they were equal. We study and strive to maximize the effectiveness of these transports in terms of computational behavior, relying on the univalence axiom as little as possible. Our approach handles both type and term dependency.

1 Introduction
If mathematics is the art of giving the same name to different things, programming is the art of computing the same thing with different means. That sameness notion ought to be equivalence. Unfortunately, in programming languages as well as proof assistants, the notion of sameness or equality is appallingly syntactic. In dependently-typed languages that expose two functions head and map, along with a simple correctness property: mapping a function f over the collection and then taking the first element is the same as taking the first element and then applying f to it. In Coq:

Record Lib (C : Type → ℕ → Type) :=
{ head : ∀ A : Type, A (S n) → A; map : ∀ A B (f : A → B) n, C A n → C B n; prop : ∀ n A B (f : A → B) (v : C A (S n)), head (map f v) = f (head v) }.

We can implement such a library using standard size-indexed vectors:

Definition libvec : Lib Vector. := (| head := Vector.hd; map := Vector.map; prop := libvec_prop |).

where libvec_prop is the proof of prop, relating the specific head and map functions on vectors.

Assuming a type equivalence between indexed vectors and standard polymorphic lists refined with a predicate on their length, univalence supports the automatic construction of an equivalent library that operates on lists, together with the same correctness property.

In a hypothetical univalent system with a univalent transport operator, hereafter noted ↑, this new library could simply be obtained as follows:

Definition liblist : Lib (fun A n ⇒ |l: list A & length l = n|) := ↑ libvec.
This way, the user gets a library on lists that is usable out of the box, and correct by construction. In particular, the proof of prop has been automatically converted to establish the property over lists. This desirable scenario is however unsupported by existing approaches.

Univalence and computation. The scenario above assumes that univalent transport is effective: given a closed term of type \( P \mathbin{\mathsf{A}} \), transport yields a closed term of type \( P \mathbin{\mathsf{B}} \). However, in the Calculus of Inductive Constructions (CIC) or Martin-Löf Type Theory, univalence is expressed as an axiom \([23]\). The univalence axiom can be used in particular to establish what we hereby call the Indiscernibility of Equivalents \(^1\), formally that \( A \cong B \) implies that \( P \mathbin{\mathsf{A}} \cong P \mathbin{\mathsf{B}} \), for any type constructor \( P \). However, by the Curry-Howard correspondence, axioms have no computational content, since they correspond to free variables. Therefore an axiomatic general univalent transport is not effective. In concrete terms, this means that using axiomatic univalent transport will yield a “stuck term”, stuck at the use of the axiom.

Since the advent of HoTT and the univalence axiom, several attempts have been made to build a dependent type theory with a computational account of univalence, most notably with work on cubical type theory (CubicalTT) \([1, 10]\). Such an approach aims at making univalence an inherent, universal property of the system, i.e. demanding that all constructions of the type theory be compatible with univalence.

Complementary to such a “clean slate” approach, there is much to gain in studying how to address the computational effectiveness of univalent transport while staying within CIC. In particular, this would allow existing proof assistants such as Coq to directly benefit from any progress in this regard, while contributing to the research question of the computational content of univalence.

Contributions. The main contribution of this work is to recognize that, while univalence cannot be generally given computational content in CIC, we can support effective univalent transport for a very large subset of CIC terms, covering most practical needs when considering programming activities. Rather than considering univalence as a universal property, we describe univalence as an ad-hoc property of the type constructors of the theory, defined as a strengthening of parametricity \([21]\), which we coin \emph{univalent parametricity}. By supporting the justification of univalent parametricity per type constructors, we can, on a case-by-case basis, try to avoid using axioms altogether, or at least push the use of axioms out of the computationally-relevant parts, hence supporting effective univalent transport for a large class of programs. More precisely:

\[ \text{• We introduce univalent parametricity as a strengthening of parametricity to ensure preservation of equivalences (Section 3).} \]

\[ \text{• We provide a logical relation for univalent parametricity defined over type constructors (Section 3.1). The principle of indiscernibility of equivalents for a type constructor amounts to the fundamental property of this logical relation (Section 3.2). We prove that each type constructor of the Calculus of Constructions with universes \( \mathbb{CC}_{\omega} \) is univalently parametric, identifying in each case the necessary assumptions (Section 3.3).} \]

\[ \text{• We also define univalent parametricity through a translation in the style of Bernardy \emph{et al.} \([7]\), which allows us to prove an abstraction theorem that entails that all terms of \( \mathbb{CC}_{\omega} \) are univalently parametric (Section 3.4).} \]

\[ \text{• The logical relation for univalent parametricity serves as the foundation for an ad-hoc realization of univalent parametricity in Coq with type classes \([22]\), which is readily applicable to existing Coq developments, such as our introductory example (Section 4).} \]

\[ \text{• We discuss the impact of the proofs of univalent parametricity on the effectiveness and efficiency of the induced univalent transport (Section 5).} \]

The technical content of this work is fully formalized and proven in Coq (v8.6), including the translation and its properties, the type class framework and its instances, as well as the examples. The Coq source files are available online at: https://coq hott.github.io/univalent parametricity/.

Section 2 provides more precise background on type equivalence, univalence and parametricity in the context of dependent type theories. Section 6 discusses related work and Section 7 concludes.

### 2 Type Equivalence, Univalence, and Parametricity

We briefly review the notions of type equivalence, univalence, and parametricity in the context of dependent type theories, highlighting the challenges that lead us to the notion of univalent parametricity.

#### 2.1 Type equivalence

A function \( f : A \rightarrow B \) is an equivalence iff there exists a function \( g : B \rightarrow A \) together with proofs that \( f \) and \( g \) are inverse of each other. More precisely, the section property states that \( \forall a : A, g(f(a)) = a \), and the retraction property dually states that \( \forall b : B, f(g(b)) = b \). An additional condition between the section and the retraction, called here the adjunction condition, expresses that the equivalence is uniquely determined by the function \( f \)—and hence that being an equivalence is proof irrelevant.

**Definition 2.1** (Type equivalence). Two types \( A \) and \( B \) are equivalent, noted \( \equiv \), iff there exists a function \( f : A \rightarrow B \) that is an equivalence.
A type equivalence therefore consists of two transport functions (i.e. \( f \) and \( g \)), as well as three properties. The transport functions are obviously computationally relevant, because they actually construct values of one type based on values of the other type. Note that from a computational point of view, there might be different ways to witness the equivalence between two types, which would yield different transports.

Armed with a type equivalence \( A \simeq B \), one can therefore manually port a library that uses \( A \) to a library that uses \( B \), by using the \( A \to B \) function in covariant positions and the \( B \to A \) function in contravariant positions. However, with type dependencies, all uses of transport at the value level can leak at the type level, requiring the use of sections or retractions to deal with type mismatches. As a result, transporting even a simple library like the one presented in Section 1 quickly yields to disaster; one desperately wishes for an automatic, general transport mechanism.

This also means that while the properties of equivalence are not used computationally for rewriting from \( A \) to \( B \) or vice versa, their computational content can matter when one wants to exploit the equivalence of constructors that are indexed by \( A \) or by \( B \). For instance, to establish that a term of type \( T \ (g(f(a))) \) actually has type \( T \ a \), one needs to rewrite the term using the section of the equivalence—which means applying it as a (computationally-relevant) function.

2.2 Univalence

The (seemingly) magical potion for automatic transport is univalence.

**Definition 2.2 (Univalence).** For any two types \( A, B \), the canonical map \( (A = B) \to (A \simeq B) \) is an equivalence.

In particular, this means that \( (A = B) \approx (A \simeq B) \). By aligning type equivalence with propositional equality, univalence allows us to generalize Leibniz’s principle of indiscernibility of identicals, to what we call the principle of Indiscernibility of Equivalents.

**Theorem 2.3** (Indiscernibility of Equivalents). For any \( P : \text{Type} \to \text{Type} \), and any two types \( A \) and \( B \) such that \( A \simeq B \), we have \( P\ A \approx P\ B \).

**Proof:** Direct using univalence: \( A \approx B \implies A = B \implies P\ A = P\ B \implies P\ A \approx P\ B \) [QED]

In particular, univalence promises immediate transport for all. If \( A \) and \( B \) are equivalent, then we can always convert some \( P \) to some (equivalent) \( P' \), i.e.:

**Corollary 2.4** (Univalent Transport). For any \( P : \text{Type} \to \text{Type} \), and any two types \( A \) and \( B \) such that \( A \simeq B \), there exists a function \( \uparrow : P\ A \to P\ B \).

There is a catch, however. Formally, univalence cannot be defined constructively in CIC and is therefore defined as an axiom. Because the proof of Theorem 2.3 starts by using the univalence axiom to replace type equivalence with propositional equality, before proceeding trivially with rewriting, it has no computational content, and hence we cannot exploit (axiomatic) univalence to reap the benefits of automatic transport of programs and their properties across equivalent types. It is important for transport to be effective, i.e. that it has computational content.

Intuitively, an effective function ensures canonicity: it never gets stuck due to the use of an axiom. Conversely, a function that uses an axiom and hence “does not compute” is called ineffective. By extension, a type equivalence \( A \simeq B \) consisting of two functions \( f : A \to B \) and \( g : B \to A \) is said to be effective iff both \( f \) and \( g \) are effective functions.

2.3 Towards effective univalent transport

HoTT and univalence advocate that type equivalence is the adequate semantic notion of equality on types. As we have seen, from a practical point of view, we want type constructors to preserve equivalences and we want to establish such a compatibility in a constructive manner so as to obtain an automatic transport that is effective.

As a matter of fact, it is feasible to prove, without using the univalence axiom, that many type constructors preserve equivalences. For instance it is not hard to prove effectively that if \( A \simeq B \), then \( \text{List} \ A \simeq \text{List} \ B \). The HoTT library for Coq does provide such compatibility lemmas for many type constructors [5]. For instance, for the dependent function and pair types, the following lemmas are proven:

**Definition equiv_functor_V : \forall A B (P : \text{Type} \to \text{Type}) (Q : B \to \text{Type})
\( (e : A \simeq B) \implies (e' : \forall P, (\forall a, P a) \approx (\forall b, Q b)) \)**

**Definition equiv_functor_Σ : \forall A B (P : \text{Type} \to \text{Type}) (Q : B \to \text{Type})
\( (e : A \simeq B) \implies (e' : \forall P, (\forall a, P a) \approx (\forall a, Q a)) \)**

Such lemmas are sufficient to automatically derive an effective definition of the head function that operates on lists—length given the head function on vectors. However, they are not really sufficient to deal with more complex dependencies. The source of the problem is that the above lemmas necessarily use transport explicitly in order to be able to state their equivalence premises (observe the type of \( e' \) in the definitions above).

To illustrate the issue, consider the Lib record type from Section 1, for which we want to prove:

\( \text{Lib Vector} \ t \defeq \text{Lib} \ (\text{fun } n \Rightarrow [\text{List} \ A \& \text{length} \ 1 \ = \ n]) \)

Recall that records are simply nested dependent pairs. By exploiting the functoriality of the dependent function and pair types with respect to equivalence, \( \text{equiv_functor_Σ} \) and \( \text{equiv_functor_V} \), for the property prop relating head and map, the transports cascade and we end up having to prove the following goal:

\( \forall n A B (f : A \to B) (l : [\text{List} \ A \& \text{length} \ 1 \ = \ n]),
(\text{head} (\text{map} \ f \ l)) = (\text{head} f \ (\text{head} l)) \)
It is now natural to try to apply the functoriality of propositional equality, defined as:

Definition equiv_eq : ∀ A B (e : A = B) (x y : A), (x y) ≜ (x = y).

However, because of all the occurrences of transport, our goal does not match the structure of that result. We first need to apply lemmas regarding the commutativity of transport in order to massage the goal such that it has the proper shape to apply equiv_eq. More generally, because of their use of transport in premises, applying the functoriality lemmas from the HoTT library yields an abundance of occurrences of transport in hard-to-predict places. This implies potentially costly back-and-forth conversions that could be avoided, and makes full automation very hard, if not impossible. Therefore, while the HoTT library shows that it is possible to obtain effective transport, the approach does not scale up to automation because of "the transport hell".

Escaping the transport hell. Looking back at the functoriality lemmas equiv_func_v and equiv_func_x, we observe that the difficulty arises because one cannot directly relate the indexed types P and Q. This is because a) they have different types, namely A → Type and B → Type, and b) type equivalence is only defined at Type. This forces the premises of these lemmas (e) to be stated extensionally, using transport on one (arbitrary!) side so that the types match.

This analysis tells us that using an heterogeneous relation, i.e. a relation between terms of different types, could allow us to side-step the need for explicit transport in premises and hence avoid an abundance of occurrences of transport. This is reminiscent of how McBride’s heterogeneous equality simplifies the formulation of Observational Type Theory [2]. Furthermore, we see that we need equivalence to not only be defined at Type, but at least as well at Type → Type to be able to relate type constructors à la F_C, as in our statement of Theorem 2.3. As a matter of fact, we also need the relation to be defined at A → Type in order to relate indexed types. Actually, to be able to state that an indexed type takes related inputs to related outputs, we need the relation to be defined at any type.

We are therefore looking for a uniform framework, based on a heterogeneous relation, that would provide us with a powerful reasoning principle like the abstraction theorem of parametricity [21]. With parametricity, terms that are related to themselves are relationally parametric; for functions, this means that they take related inputs to related outputs, similarly to what we are after. As we describe next, Reynolds’ notion of parametricity, extended to dependent type theories, is too weak to allow us to reason about preservation of equivalences. However, as we will develop in Section 3 and beyond, we can strengthen parametricity to provide us with "equivalences for free!".

\[ \llbracket \text{Type} \rrbracket_p A B \doteq A \rightarrow B \rightarrow \text{Type} \]
\[ \llbracket \Pi A : \text{Type} \rrbracket_p f g \doteq \Pi(a : A)(a') \langle e : \llbracket A \rrbracket_p a a' \rangle \llbracket B \rrbracket_p (f a)(g a') \]
\[ \llbracket x \rrbracket_p \doteq x_r \]
\[ \llbracket \lambda x : A. t \rrbracket_p \doteq \lambda(x : A)(x') \langle x_r, (\llbracket x \rrbracket_p x x') \rangle \llbracket t \rrbracket_p \]
\[ \llbracket \Gamma, x : A \rrbracket_p \doteq \llbracket \Gamma \rrbracket_p, x, x' : \llbracket A \rrbracket_p x x' \]

Figure 1. Parametricity translation for CC_o (from [7])

2.4 Parametricity for dependent types

Reynolds originally formulated the relational interpretation of types to establish parametricity of System F [21]. Recently, Bernardy et al. [7] generalized the approach to pure type systems, including the Calculus of Constructions with universes CC_o, and its extension with inductive types, the Calculus of Inductive Constructions CIC, which is at the core of proof assistants like Coq. This section develops the approach in sufficient details to follow our proposal.

The syntax of CC_o includes a hierarchy of universes Type_i, variables, applications, lambda expressions and dependent function types:

\[ A, B, M, N ::= Type_i | x | M N | \lambda x : A. M | \Pi x : A. B \]

Its typing rules are standard, and hence omitted here—see [20] for a recent presentation.

Parametricity for CC_o can be defined as a logical relation \[ \llbracket A \rrbracket_p \] for every type A. Specifically, \[ \llbracket A \rrbracket_p a_1 a_2 \] states that \( a_1 \) and \( a_2 \) are related at type \( A \). The essence of the approach is to express parametricity as a translation from terms to the expression of their relatedness within the same theory; indeed, the expressiveness of CC_o allows the logical relation to be stated in CC_o itself. Note that because terms and types live in the same world, \[ \llbracket \text{Type} \rrbracket_p \] is defined for every term.

Figure 1 presents the definition of \[ \llbracket \text{Type} \rrbracket_p \] for CC_o, based on the work of [7]. For the universe \( \text{Type}_i \), the translation is naturally defined as (arbitrary) binary relations on types. For the dependent function type \( \Pi a : A. B \), the translation specifies that related inputs at \( A \), as witnessed by \( e \), yield related outputs at \( B \). Note that, following [7], the prime notation (e.g. \( A' \)) denotes duplication with renaming, where each free variable \( x \) is replaced with \( x' \). Similarly, the translation of a lambda term \( \lambda x : A. t \) is a function that takes two arguments and a witness \( x_r \) that they are related; a variable \( x \) is translated to \( x_r \); a translated application passes the original argument, its renamed duplicate, along with its translation, which denotes the witness of its self-relatedness. The translation of type environments follows the same augmentation pattern, with duplication-renaming of each variable as well as the addition of the relational witness \( x_r \).
Theorem 2.5 (Abstraction theorem). If $\Gamma \vdash t : A$ then
\[
\Gamma \vdash \llbracket t \rrbracket_p : \llbracket A \rrbracket_p \times t'.
\]

In particular, this means that the translation of a term $\llbracket t \rrbracket_p$ is itself the proof that $t$ is relationally parametric.

The abstraction theorem is proven by showing the fundamental property of the logical relation for each constructor of the theory. In particular, for the cumulative hierarchy of universes, $\llbracket \text{Type}_i \rrbracket : \text{Type}_{i+1}$, this means that we have a kind of fixpoint property for the relation on $\text{Type}_i$:
\[
\lambda(A \rightarrow B) : \text{Type}_i \rightarrow \text{Type}_i \rightarrow \text{Type}_{i+1}.
\]

For parametricity, this property holds because
\[
\lambda(a : A) \cdot (b : B). a = t b,
\]
we cannot freely conclude that $A$ and $B$ are themselves equivalent; indeed, we only know that $A$ and $B$ are in relation, without any additional constraint on this relation.

Similarly, in our $	ext{Lib}$ example, we can show that $	ext{Lib}$ is related to itself, meaning it is relationally parametric, but that does not imply that it preserves equivalences.

The main conceptual contribution of this work is to precisely identify how to strengthen the parametricity relation to be able to deduce such equivalences, hence allowing automatization of effective transport.

3 Univalent Parametricity

This section develops our approach to univalent parametricity for $\text{CC}_n$.

We first define a univalent logical relation as a type-indexed logical relation on all the type constructors of $\text{CC}_n$ (Section 3.1). A term is univalently parametric if it is related to itself; in particular, we prove that univalently parametric constructors satisfies the Indiscernibility of Equivalents (Section 3.2). We discuss in Section 3.3 the proofs that each type constructor is univalently parametric, paying attention to the potential use of axioms.

\[
A = B : \text{Type}_i \rightarrow \text{Type}_i \rightarrow \text{Type}_{i+1} \neq \text{Type}_i \approx B.
\]

In words, on the left-hand side we have an arbitrary relation on $\text{Type}_i$, while on the right-hand side, we have an equivalence.

Another intuitive approach is to state that $\llbracket \text{Type}_i \rrbracket : \text{Type}_i \rightarrow \text{Type}_{i+1} \neq \text{Type}_i \approx \text{Type}_i$.

Figure 2. Univalent relation for $\text{CC}_0$
between the two notions. This in particular implies that the
identity type, which defines the notion of equality, will not
satisfy the fundamental property of the logical relation. For
this, we need to additionally demand that the heterogeneous
relation coincides with propositional equality once the values
are at the same type.

Formally, we introduce a logical relation for univalent
parametricity, called the univalent relation, defined in Fig-
ure 2 and noted \( x \approx y : X \rightsquigarrow Y \), which relates two terms \( x \) and \( y \) of possibly different types \( X \) and \( Y \), and is defined over all the type constructors of \( \text{CC}_e \). We write simply \( X \rightsquigarrow Y \) to
specify that the univalent relation is defined between \( X \) and
\( Y \), i.e. \( \cdot \approx \cdot : \to X \rightsquigarrow Y \) is defined.

At \( \text{Type}_i \), the univalent relation \( A \approx B : \text{Type}_i \rightsquigarrow \text{Type}_i \)
requires both \( A \approx B \) and \( A \approx B \), as well as a coherence
condition between the heterogeneous relation and equality.
This (crucial!) condition stipulates that the heterogeneous
relation does coinide with propositional equality up to a
transport using the equivalence, i.e.:
\[
(a \approx b : A \rightsquigarrow B) \approx (a \equiv b)
\]

Note that the use of transport on one (arbitrary) side breaks
the symmetry of the definition, in the same way as the Coq
HoTT library functoriality lemmas such as \( \text{equiv}_\text{functor}_\chi \)
do (Section 2.3). The fundamental difference is that in our
approach, this arbitrary choice is deferred as late as possible,
i.e. when we do need to know more about the univalent
relation.

As alluded to above, the coherence condition is used in
particular in the proof that the identity type is related to
itself. In that case, we need to prove that
\[
\forall A B : \text{Type}, \forall a a' : A, \forall b b' : B',
\]
\[
a \equiv b : A \Rightarrow B \wedge a' \equiv b' : A \Rightarrow B \Rightarrow a = a' \equiv b = b'
\]
which is possible only if we know that related inputs are
equal up to transport.

Consequently, to establish a univalent relation between
two types, it is not enough to exhibit an arbitrary relation;
one also needs to prove that both types are equivalent, and
that the relation satisfies the coherence condition.

On the other type constructors, the univalent logical rela-
tion is similar to parametricity. In particular, at type families
\( A \to \text{Type}_i \) and \( B \to \text{Type}_i \), the univalent relation says that
\( A \) and \( B \) must be related and that for every related input, the
applied type families must be related at \( \text{Type}_i \). In the same
way, at dependent function types \( \Pi a : A. P a \) and \( \Pi b : B. Q b \),
the univalent relation says that type families \( P \) and \( Q \) must
be related and that for every related indices \( a \) and \( b \), we get
related outputs at \( P a \) and \( Q b \).

3.2 Univalent parametricity and Indiscernibility of
Equivalents

Univalently parametric terms are those “in the diagonal” of
the univalent relation, i.e. that are related to themselves.

**Definition 3.1** (Univalent parametricity). Let \( x : X \), we say
that \( x \) is univalently parametric, or simply univalent, notation
\( \text{Univ}(x) \), if \( x \approx x : X \rightsquigarrow X \).

Using the univalent relation presented above, we cannot
establish its fundamental property (namely, that all well-
typed \( \text{CC}_e \) terms are univalently parametric); we will do
so in Section 3.4 using a translation. But we can already
state and prove an important property: that a univalently
parametric type constructor preserves type equivalences.

**Proposition 3.2** (Univalent constructor preserves equivalences). Let \( P : \text{Type}_i \to \text{Type}_i \) be a univalently parametric constructor, i.e. \( \text{Univ}(P) \), then \( A \approx B \implies P A \approx P B \).

**Proof.** If we unfold the definition, \( \text{Univ}(P) A B \) means that
\( A \approx B : \text{Type}_i \Rightarrow \text{Type}_i \), as well as a coherence
condition between the heterogeneous relation and equality.
Because we know that \( A \approx B \), we can build the canonical
heterogeneous relation
\[
\lambda(a : A)(b : B).a \equiv b
\]
which trivially satisfies the coherence condition, so \( A \approx B : \text{Type}_i \Rightarrow \text{Type}_i \). Therefore, \( P A \approx P B : \text{Type}_i \Rightarrow \text{Type}_i \)
which in particular means that \( P A \approx P B \).

**Note.** In the proof, we use the canonical relation \( \lambda(a : A)(b : B).a \equiv b \), which uses both equality and univalent
transport, to get a term \( \text{canon}(e) : A \approx B : \text{Type}_i \Rightarrow \text{Type}_i \)
from \( e : A \approx B \). One might wonder why the definition of
the univalent relation does not “hardcode” this canonical
relation, instead of allowing any heterogeneous relation that
satisfies the coherence condition. This decision is in fact
technically very important because if we eagerly imposed
the use of this relation, we would be back in transport hell
as described in Section 2.3.

3.3 Type constructors are univalently parametric

We now prove that the universe \( \text{Type}_i \) and the dependent
function type \( \Pi \) are univalent.

3.3.1 Type

\( \text{Univ}(\text{Type}_i) \) corresponds to the fixpoint property on the
universe of the logical relation, and requires the univalence
axiom to be valid in CIC.

**Proposition 3.3.** \( \text{Univ}(\text{Type}_i) \) is inhabited.

**Proof.** First, we need to define a relation between \( \text{Type}_i \) and
\( \text{Type}_i \). By a fixpoint argument, it has to be \( \text{Type}_i \Rightarrow \text{Type}_i \).
We also need to provide an equivalence \( \text{Type}_i \cong \text{Type}_i \); we
simply take the identity equivalence \( \text{id}_{\text{Type}_i} \). Finally, we need
to prove that the relation is coherent with equality, that is,
we need to exhibit a term \( \text{univ}_{\text{Type}_i} \), such that:
\[
\text{univ}_{\text{Type}_i} : \Pi A B. (A \approx B : \text{Type}_i \Rightarrow \text{Type}_i) \cong (A = B)
\]

For the function from \( A \approx B \) to \( A \approx B : \text{Type}_i \Rightarrow \text{Type}_i \), by
induction on equality, it is sufficient to provide the canonical

We now show that the dependent function type is univalently equivalent to the identity equivalence, as used in the proof of Proposition 3.2.

The rest of the proof makes use of univalence and in particular the section and the retraction of the equivalence postulated by the univalence axiom, together with lemmas about decomposition of equality over Type_i ≔ Type_i and commutation of transports; the interested reader can consult the Coq development.

\[\text{3.3.2 Prop}
\]

In our definition, Prop is treated in the same way as Type_i, because Prop : Type_i is a universe also enjoying the univalence axiom. The only specificity of Prop is its impredicativity, which does not play a role here.

**Proposition 3.4.** Univ(Prop) is inhabited.

**Proof.** Special case of the fact that Univ(Type_i) is inhabited.

It is also possible to state a stronger axiom on Prop called propositional extensionality, which uses logical equivalences instead of type equivalences in its statement:

\[\text{(P ≡ Q) ≔ (P \iff Q),}\]

This axiom can be derived from univalence and proof irrelevancy for Prop but is stronger than just univalence. As we are looking for the minimal amount of axioms needed for establishing univalent parametricity, we do not make use of this stronger axiom.

Note that exploiting the fact Prop is proof irrelevant, Prop ≔ Prop boils down to

\[A ≈ B \land A \iff B \iff \forall a : A, b : B, \text{IsContr}(a \approx b : A \approx B),\]

where IsContr A says that A is contractible, i.e. has a unique inhabitant. This is because for all a and b, the type \(\text{a \approx b : A \approx B}\) is contractible and being equivalent to a contractible type is the same as being contractible. The definition we obtain in this case coincides with the definition of parametricity with uniformity of propositions, recently developed by Anand and Morrisset [3]—more details in Section 6.

\[\text{3.3.3 Dependent function type}\]

We now show that the dependent function type is univalently parametric. This result requires functional extensionality, i.e. the fact that the canonical map

\[f = g : \Pi(x : A). f x = g x\]

is an equivalence. This property is a consequence of univalence.

**Proposition 3.5.** Univ(Π) is inhabited.

---

\[\text{Submission to CPP 2018, January 8-9, Los Angeles, CA, USA}\]

\[\text{Proof.} \ \text{Univ}(\Pi) A B P Q \text{ unfolds to}\]

\[A \approx B : \text{Type}_i \iff \text{Type}_i \rightarrow P \approx Q : A \rightarrow \text{Type}_i \iff B \rightarrow \text{Type}_i \rightarrow \Pi(a : A). P a \approx \Pi(b : B). Q b : \text{Type}_i \iff \text{Type}_i\]

First, we need to define a relation between \(\Pi(a : A). P a \approx \Pi(b : B). Q b\). This is of course the definition of \(\Pi(a : A). P a \iff \Pi(b : B). Q b\) as given in Figure 2.

Next, we need to show that \(\Pi(a : A). P a \iff \Pi(b : B). Q b\) knowing that \(A \approx B\) and \(\Pi(a : A)(b : B).a \approx b : A \approx B \rightarrow P a \approx Q b\). Using the equivalence between \(a \approx b : A \approx B\) and \(a =?b\), this boils down to \(\Pi(a : A). P a \approx Q(\tau a)\).

At this point we can apply a standard result of HoTT [23], namely equiv_function in the Coq HoTT library [5], which was already introduced in Section 2.3. This lemma is effective for the two transport functions but requires functional extensionality in the proof that they form an equivalence.

We note the resulting term Equiv_{Π}, with:

\[\text{Equiv}_{Π} : ΠA B P Q. A \approx B \rightarrow (\Pi(a : A)(b : B).a \approx b : A \iff B \rightarrow P a \approx Q b) \rightarrow Π(a : A). P a \iff Π(b : B). Q b.\]

The proof that the relation is coherent with equality is the novel part required by univalent parametricity. This means that we need to define a term

\[\text{univ}_{Π} : Πf g.(f \equiv g : Πa : A. P a \iff Πb : B. Q b) \equiv (f = ?g)\]

This part is quite involved as it is exactly where we show that transporting in many hard-to-predict places is equivalent to transporting only at the top level, thereby avoiding the transport hell described in Section 2.3. Again, the interested reader can consult the Coq development.

\[\text{3.4 Univalent parametricity translation}\]

Proving the general fundamental theorem of univalent parametricity requires an induction on the whole syntax of CC_unib, including variables, application and lambda expressions, and is therefore better handled by a translation in the style of Bernardy et al. (recall Figure 1 of Section 2.4). Figure 3 shows how to extend the relational parametricity translation to force the heterogeneous relation defined between two types to correspond to a type equivalence with the coherence condition. Note that the translation does not target CC_u, but rather CIC_u, which is CIC plus the univalence axiom. We note \(Γ \vdash u.t : T\) to stipulate that the term is typeable in CIC_u.

The definition of the translation of a type A is more complex than that of Figure 1 because in addition to the relation \([A]_u\), we need an equivalence \([A]_{u \text{eq}}\) and a witness \([A]_{coh}\) that the relation is coherent with equality.

\[\text{The definition of the inverse function requires using the retraction, and the proof that it forms a proper equivalence requires the adjunction condition (Section 2.1). This means that the dependent function type would not be univalent if we replaced type equivalence with a simpler notion, such as the possibility to go from one type to another and back, or even by isomorphisms.}\]
they make implicit use of $[A]_{u}^{coh}$, which explains why this part of the translation is not directly visible in Figure 3.

For the other terms, the translation does not change with respect to parametricity except that $[\_\_]_{u}$ must be used accordingly when we are denoting the relation induced by the translation and not the translation itself.

We can now derive the abstraction theorem of univalent parametricity.

**Theorem 3.6 (Abstraction theorem).** If $\Gamma \vdash t : A$ then $[\Gamma \vdash t : A]_{u} r_{u} [\Gamma]:[A]_{u} t'$.

**Proof.** The proof is a straightforward induction on the typing derivation. The interested reader can consult the Coq development.

Actually, we are more interested in the corollary that states that every term of $CC_{co}$ is univalently parametric.

**Corollary 3.7 (Fundamental property).** If $\Gamma \vdash a : A$ then $Univ(a)$.

**Proof.** For a closed term, we have $[A]_{u} a \approx A$ and $a' \approx a$, so by the abstraction theorem, $[\_\_]_{u} a a : A \approx A$. □

Finally, note that although the translation for dependent function types is defined for two terms $a$ and $a'$ of respective types $A$ and $A'$, $A'$ is not any arbitrary type: it is the result of duplication with renaming applied to $A$ (Section 2.4); likewise, $a'$ is a renamed duplicate of $a$. Additionally, $a$ and $a'$ are expected to be related according to the interpretation of the single type $A$. This is why the univalent logical relation of Figure 2 is more general than the univalent parametricity translation: it can describe relations between terms of arbitrarily different types, as long as some equivalence can be exhibited. For instance, we can relate naturals $\mathbb{N}$ and binary naturals $\mathbb{N}$, i.e., $\mathbb{N} \approx \mathbb{N} : Type$.

### 3.5 Dependent pairs

It is possible to extend the translation to inductive types of CIC. While we leave a general treatment of inductive types to future work, in this paper we deal with dependent pairs types, and with record types (as nested dependent pair types). The Coq formalization also deals with other inductive types such as the identity type and abstract data types.

In CIC, dependent pairs are defined as the inductive family:

\[
\text{Inductive sigT} (A : Type) (B : A \rightarrow Type) : Type := \\
\text{existT} : \forall x : A, B x \rightarrow \{x : A \& B x\}.
\]

Thus, the unique constructor of a dependent pair is existT and the elimination principle is given by

\[
\text{sigT_rect} \; (A : Type) (P : A \rightarrow Type) (P_0 : \text{sigT} A P \rightarrow Type), \; (x (a x p) : P x p) \rightarrow \forall s : \text{sigT} A P, P s.
\]

As common, we use the notation $\Sigma a : A. B$ to denote sigT A (fun a ⇒ B), similarly to dependent type theories where pair types are part of the syntax [19].
Figure 4. Univalent relation for dependent pairs

The univalent relation between $\Sigma a : A. P a$ and $\Sigma b : B. Q b$ is defined in Figure 4. It naturally requires the type families $P$ and $Q$, as well as the first and second elements of the pair, to be related at the corresponding types.

Proposition 3.8. \textit{Univ(2) is inhabited.}

Note that the last equivalence used in the proof, namely that

$$(\Sigma p : x.1 \Rightarrow y.1 . x.2 = y.2) \Rightarrow (x \Rightarrow y)$$

is the counterpart of functional extensionality for dependent function types. The main difference is that this equivalence is effective as it can be proven by elimination of dependent pairs.

The proofs that the constructor $\text{existT}$ and the eliminator $\text{sigT}_\text{rect}$ are univalently parametric are direct by induction on the structure of a dependent pair type.

4 Univalent Parametricity in Coq

The whole development of univalent parametricity exposed in this article has been formalized in the Coq system [12], reusing several constructions from the HoTT library [5]. We do not discuss the Coq formalization of the univalent parametricity translation of Section 3.4 here; instead, we focus on the shallow embedding of the univalent relation based on type class instances to define and automatically derive the univalence proofs of Coq constructions. We first introduce the core classes of the framework in Section 4.1, and then describe the instances for some type constructors in Section 4.2.

4.1 Coq framework

The central notion at the heart of this work is that of type equivalences, which we formulate as a type class to allow automatic inference of equivalences:

\begin{verbatim}
Class Equiv A B := \{ e_fun := A -> B ; e_isequiv := isEquiv e_fun \}.
\end{verbatim}

Notation "A ⇔ B" := (Equiv A B).

This way, we can define automatic transport as

\begin{verbatim}
Definition univalent_transport [A : Type] [e : A → B] : A → B := e_fun e.
\end{verbatim}

Notation "\text{univalent_transport}" := univalent_transport.

where the equivalence is obtained through type class instance resolution, i.e. proof search.

4.2 Univalent type constructors

The core of the development is devoted to the proofs that standard type constructors are univalently parametric, notably Type and I. In terms of the Coq framework, this means providing $\text{UR_Type}$ instances relating each constructor to itself. These instance definitions follow directly the proofs discussed in Section 3.

For the universe $\text{Type}_i$, we define:

\begin{verbatim}
Instance UR_Type def[@i j] : UR@[@i j j] Type@[i] := \{ \text{ur} := UR_Type@[i i i] \}.
\end{verbatim}

This is where our fixpoint construction appears: the relation at $\text{Type}_i$ is defined to be $\text{UR_Type}$ itself. So, for a type to be in the relation means more than mere equivalence: we also get a relation between elements of that type that is coherent with equality. This $\text{UR_Type def}$ instance will be used implicitly everywhere we use the notation $X \approx Y$, when $X$ and $Y$ are types themselves. Thanks to the implicit cumulativity of universes in Coq, we do not need to worry about lifting our constructions from lower to larger types in general, so from now on we will omit the universe annotations (like $\forall (j j j)$ above), although some annotations appear in the Coq source files in order to explicitly validate our assumptions about universes.

For dependent function types, we set:

\begin{verbatim}
Definition UR_Forall A A' (B : A → Type) (B' : A' → Type) (dom : UR A A') (codom : ∀ x y (H : x = y), UR (B x) (B' y)) : UR (∀ x : B) (∀ y, B' y) := \{ \text{ur} := fun f g ⇒ ∀ x y (H : x = y), f x ≃ g y \}.
\end{verbatim}

The univalent parametricity relation on dependent function types expects relations on the domain and codomain types, the latter being parameterized by the former through
5.1 Effectiveness

Let us go back to the library example from Section 1.

Because we are in an (axiomatic) univalent theory, we can always immediately appeal to the univalence axiom and functional extensionality to establish that a given constructor is univalently parametric by resorting to univalence:

\[
\text{Definition } \text{FP}_\text{Lib}: \text{Lib} = \text{Lib}.
\]

It produces equality based on a pointwise equivalence. However, using this proof of the fundamental property of the univalent relation for \(\text{Lib}\) yields a transport that is not effective. To get effectiveness, we need a direct proof, using automation for record types:

\[
\text{Definition } \text{FP}_\text{Lib}: \text{Lib} = \text{Lib}.
\]

As explained above, this involves an equivalence with the nested pair type version of the interface, plus a proof search showing that each of the types in the interface is related to itself by \(\approx\).

We can then start from an implementation of the library on vectors, such as:

\[
\text{Definition libvec : Vector.t} := \\
\text{ | } \text{head} := \text{fun A n x} ⇒ @\text{Vector.hd} A n x; \\
\text{ | } \text{map} := \text{fun A B f n} ⇒ \text{Vector.map f;} \\
\text{ | } \text{prop} := \text{lib_vector_prop}.
\]

We then need to define that indexed vectors and lists refined with their length are related by the univalent relation (note the importance of the heterogeneity of the relation for user-defined applications).

Deriving the implementation for the equivalent type of lists of fixed length simply amounts to an application of transport, as wished for in the introduction of this paper:

\[
\text{Definition liblist : Lib (fun A n} ⇒ \{l : list A & length l = n\}) := \\
\text{libvec}.
\]

We can check the effectiveness of this construction on a simple example:\(^8\)

\[
\text{Eval compute in liblist(map)$ [[[1;2]]].}
\]

\[
\text{=} [[2;3]]
\]

\[
\text{;}[l : \text{list} N & \text{length} l = 2]
\]

And indeed, the correctness property has automatically been ported to lists:

\[
\text{Check lib_list.(prop _.).}
\]

\[
\text{;} \forall n : \text{list} A \rightarrow B \forall n : \{l : \text{list} A & \text{length} l = S n\}, \\
\text{head lib_list.(map lib_list f v) = f (head lib_list v)}
\]

\(^8\)The notation \([X;\ldots;Z]\) creates a list with a proof (by reflexivity) that it is of the appropriate size.
5.2 Efficiency

Another possible application of equivalences for free is the use of transport to switch between easy-to-reason-about and efficient representations, an approach known as data refinement [11]. For instance, it is possible to show that (ordinary) natural numbers \( \mathbb{N} \) and binary natural numbers \( \mathbb{N}_b \) are univalently related.

We can then exploit this relation to automatically define the power function on \( \mathbb{N} \) by transporting the (efficient) power function on \( \mathbb{N}_b \):

**Definition** \( \texttt{N\_pow} : \mathbb{N} \to \mathbb{N} \to \mathbb{N} = \uparrow \texttt{N\_pow} \).

Evaluating \( \texttt{N\_pow} \texttt{2} \texttt{26} \) takes around 20 seconds while computing it with the standard power function directly defined on \( \mathbb{N} \) takes twice that time. This illustrates that the cost of transporting from one representation to another can be balanced when the computation involved is much more efficient on one side.

But effective proofs of univalent parametricity are not equivalent in practice. For instance, we could define univalence of \( \texttt{Vect} \ A \) \( n \) through the equivalence with lists refined with a predicate on their length and the fact that lists refined with a predicate on their length are univalent. While the induced transport is effective, it is far from optimal computationally because it implies going through lists, which means creating intermediate data structures.

For instance, suppose a (not so large) vector \( \texttt{largeVector} : \texttt{Vector.t \mathbb{N} 20} \) of size \( 20 \) has been defined. Using our framework, it is possible to define automatically

**Definition** \( \texttt{largeVectorN} : \texttt{Vector.t \mathbb{N} 20} := \uparrow \texttt{largeVector} \).

However, evaluating (the compiled version of) \( \texttt{largeVectorN} \) takes around 3 seconds on a quad-core Intel Core i5 3.5GHz machine.

If instead we directly define the equivalence using a direct reasoning on vectors, the evaluation of the same term obtained by transporting along this equivalence takes less than 1 millisecond! Likewise, transporting functions that operate on vectors of \( \mathbb{N} \) to functions that operate on vectors of \( \mathbb{N}_b \) will be much more efficient if the direct equivalence is used.

In summary, we highlight the importance of the open-endedness of the type class framework, allowing programmers to define instances that are tailored to their specific needs in terms of effectiveness and efficiency, to be balanced with the complexity of the proof to provide.

6 Related Work

**Type theories.** Homotopy Type Theory [23], and its embodiment in the HoTT library [5] treat equality of types as equivalence. For regular datatypes (also know as homotopy sets or hSets), equivalence boils down to isomorphism, hence the existence of transports between the types. However, as univalence is considered as an axiom, any meaningful use of the equality type to transport terms along equivalences results in the use of a non-computational construction. In contrast we carefully delimit the effective equivalence preserving type constructors in our setting, pushing axioms as far as possible.

Cubical Type Theory [10] provides computational content to the univalence axiom, and hence functional and propositional extensionality as well. In this case, the invariance of constructions by type equivalence is built in the system and the equality type reflects it. Note that the recent work of Altenkirch and Kaposi on a cubical type theory without an interval [1] proposes a similar use of an heterogeneous relation but in our framework, we relate the heterogeneous relation to equality, which allows us to stay within CIC, without relying on another type theory.

Observational Type Theory (OTT, [2]) uses a different notion of equality, coined John Major equality. It is a heterogeneous relation, allowing to compare terms in potentially different types, usually with the assumption that the two types will eventually be structurally equal, not merely equivalent. This stronger notion of equality of types is baked in the type system, where type equality is defined by recursion on the type’s structure, and value equality follows it. It implies the K axiom which is in general inconsistent with univalence, although certainly provable for all the non polymorphic types definable in OTT. A system similar to ours could be defined on top of OTT to allow transporting by equivalences.

Parametric Type Theory and the line of work integrating parametricity theory to dependant type theory, either internally [6] or externally, is linked to the current work in the sense that our univalent parametricity translation is a refinement of the usual parametricity translation. We however do not attempt to make the theory internally univalent as we recognized that not all constructions in CIC are effectively univalent.

For Extensical Type Theory, Krishnaswami and Dreyer [15] develop an alternative view on parametricity, more in the style of Reynolds, by giving a parametric model of the theory using quasi-PERs and a realizability interpretation of the theory. From this model construction and proof of the fundamental lemma they can justify adding axioms to the theory that witness strong parametricity results, even on open terms. However they lose the computability and effectiveness of Bernardy’s construction or ours.

The parametricity translation of Anand and Morrisset [3] extends the logical relation at propositions to force that related propositions are logically equivalent. It can be seen as a degenerate case of our extension which forces related types to be equivalent, as equivalence boils down to logical equivalence on propositions (see Section 3.3 for a more detailed explanation). However the translations differ in other aspects. While our translation requires the univalence axiom, theirs assumes proof irrelevance and the K axiom, and does...
not treat the type hierarchy. Our solution to the fixpoint arising from interpreting $\text{Type}_{i+1}$ is original, along with the use of conditions to ensure coherence with equality. They study the translation of inductively-defined types and propositions in detail, giving specific translations in these two cases to accommodate the elimination restrictions on propositions, and are more fine-grained in the assumptions necessary on relations in parametricity theorems. In both cases, the constructions were analyzed to ensure that axioms were only used in the non-computational parts of the translation, hence they are effective.

### Data refinement.

Another part of the literature deals with the general data refinement problem, e.g., the ability to use different related data structures for different purposes: typically simplicity of proofs versus efficient computation. The frameworks provide means to systematically transport results from one type to the other.

Magaud and Bertot [17, 18] first explored the idea of transporting proof terms from one data representation to another in Coq, assuming the user gave a translation of the definitions from one datatype to the other. It is limited to isomorphism and implemented externally as a plugin. The technique is rather invasive in the sense that it supports the transport of proof terms that use the computational content of the first type (e.g., the reduction rules for $\text{plus}$ on natural numbers) by making type conversions explicit, turning them into propositional rewrite rules. This approach breaks down in presence of type dependency.

In CoqEAL [11] refinement is allowed from proof-oriented data types to efficiency-oriented ones, relying on generic programming for the computational part and automating the transport of theorems and proofs. They not only deal with isomorphisms, but also quotients, and even partial quotients, which we cannot handle. Still, they can and do exploit parametricity for generating proofs but they do not support general dependent types, only parametric polymorphism.

Haftmann et al. [13] explain how the Isabelle/HOL code generator uses data refinements to generate executable versions of abstract programs. The refinement relation used is similar to the partial quotients of CoqEAL. The Autoref tool for Isabelle [16] also uses parametricity for refinement-based development. It is an external tool to synthesize executable instances of generic algorithms and refinement proofs.

Huffman and Kunčar [14] address the problem of transferring propositions between different types, typically a representation type (e.g., integers) to an abstract type (e.g., natural numbers) in the context of Isabelle/HOL. Again this allows to relate a type and its quotient, like in CoqEAL, and is based on parametricity. Recently, Zimmermann and Herbelin [25] present an algorithm and plugin to transport theorems along isomorphisms in Coq similar to that of Huffman and Kunčar [14]. In addition to requiring the user to provide a surjective function $f$ to relate two data types, their technique demands that the user explicitly provide transfer lemmas of the form $\forall x_1 \ldots x_n, R(x_1 \ldots x_n) \implies R'(f(x_1) \ldots f(x_n))$, for each relation $R$ that the user expects to transfer to a relation $R'$. The approach is not yet able to handle parameterized types, let alone dependent ones.

### 7 Conclusion

This work explores an approach to maximize the computational content of univalence in a dependent type theory. To this end, we develop the notion of univalent parametricity, which strengthens the parametricity theory of dependent type theory to ensure preservation of equivalences. We introduce an heterogeneous univalent parametricity relation and translation for $\text{CC}_\omega$ based on it. The proofs of univalent parametricity of type constructors are computationally relevant because they induce the function that allows to transport definitions and proofs over a given type to equivalent definitions and proofs over an equivalent type.

Because we need to rely on the univalence (and function extensionality) axiom in few places, effectiveness is still an issue, but we have been very careful to push axioms as far back as possible to get maximal effectiveness of the basic equivalences we provide. In practice, this means that our Coq framework can readily be used to transport certified libraries and theories along type equivalences.

While giving a computational interpretation of univalence for inductive types in full generality, if at all possible, is still an open research question, our work contributes by providing an extensible framework on top of which to experiment. Indeed, the open-ended nature of the type class framework allows users to extend and improve on our current definitions, for instance by providing more effective or more efficient instances for specific type constructors.

### Acknowledgments

We thank Pierre-Évariste Dagand and Eric Finster for useful comments/suggestions and Simon Boulier and Gaëtan Gilbert for some parts of the Coq source code.

### References


