Equivalences for Free!

Nicolas Tabareau, Éric Tanter, Matthieu Sozeau

Equivalences for Free!

Univalent Parametricity for Effective Transport

NICOLAS TABAREAU, Gallinette Project-Team, Inria, France
ÉRIC TANTER, Pleiad lab, DCC - University of Chile, Chile
MATTHIEU SOZEAU, Pi.R2 Project-Team, Inria and IRIF, France

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 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. Our approach handles both type and term dependency. We study how to maximize the effectiveness of these transports in terms of computational behavior, and identify a fragment useful for certified programming on which univalent transport is guaranteed to be effective.

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 also serve as proof assistants, equivalences can be stated and manually exploited, but they cannot be used as transparently and conveniently as syntactic or propositional equality. The benefits we ought to get from having equivalence as the primary notion of sameness include the possibility to state and prove results about a data structure (or mathematical object) that is convenient to formally reason about, and then automatically transport these results to other structures, for instance ones that are computationally more efficient, albeit less convenient to reason about. Since the seminal work of Magaud and Bertot [19] on translating proofs between different representations of natural numbers in Coq, there has been a lot of work in this direction, motivated by both program verification and mechanized mathematics, with several libraries available for either Isabelle/HOL [15] or Coq [11, 27]. At their core, most of these approaches build on parametricity [23] and its potential for free theorems [26] in order to obtain results such as data refinements for free [11] and proofs for free [7]. Despite these advances, exploiting equivalences between data structures in order to automatically transport programs, theorems and proofs, remains an elusive objective. One of the reasons, as we will demonstrate, is that parametricity is not strong enough to ensure preservation of equivalences.

Univalence [25] is a new foundation for mathematics and type theory that postulates that equivalence is equivalent to equality. Leaving aside the most profound mathematical implications of Homotopy Type Theory (HoTT) and univalence, these new foundations should fulfill the promise of automatic transport of programs, theorems, and proofs across equivalences. It should be possible to transport a library that operates over a given type A to an *equivalent* library that works with an *equivalent* type B, along with all its correctness guarantees.

Authors' addresses: Nicolas Tabareau, Gallinette Project-Team, Inria, Nantes, France; Éric Tanter, Pleiad lab, DCC - University of Chile, Santiago, Chile; Matthieu Sozeau, Pi.R2 Project-Team, Inria and IRIF, Paris, France.

2018. XXXX-XXXX/2018/3-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Univalent transport in action. We illustrate the practical benefits of univalent transport with a number of scenarios, which are not within reach of existing approaches. These examples are all supported in our Coq library, which provides a univalent transport operator, hereafter noted ↑.

Consider the polymorphic signature of a size-indexed collection data type that exposes 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} {n : ℕ}, C 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 collection library using standard size-indexed vectors:

```
Definition libvec : Lib Vector.t := {| 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. With our Coq library, this new collection 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.

Another application consists in using univalent 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 ℕ and binary natural numbers `N` are equivalent. We can then exploit this relation to automatically define the power function on ℕ by transporting the (efficient) power function on `N`:

```
Definition ℕ_pow : ℕ → ℕ → ℕ := ↑ N.pow.
```

With our library, evaluating `ℕ_pow 2 26` takes half the time of computing it with the standard power function directly. 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.

From a software engineering point of view, univalent transport can also prove particularly helpful whenever a sudden change in the representation of some part of a development seems required. Suppose that in the middle of a large development that initially depends on the inductive version of natural numbers, ℕ, the programmer is faced with a stack overflow error in Coq that is traced back to the use of ℕ. In the current state of affairs, the most direct solution is to switch from ℕ to binary natural numbers `N` from the very beginning of the definitions. This unfortunately breaks all the development, and the programmer then has to manually adapt all the definitions and proofs that eventually rely on ℕ in order to accommodate the binary representation. Alternatively, one can refactor the development to use an abstract interface for numbers and apply the data refinement

approach of CoqEAL [11]. This has the benefit of eventually being robust with respect to potential future changes. But in both cases, a complete refactoring is needed.

Conversely, univalent transport allows the programmer to automatically lift all the on-going development to use N, both in computationally-relevant parts and in parts that deal with reasoning and formal properties, without any further manual intervention. Eventually, the programmer might decide to proceed with a global refactoring of some sort, but she does not need to do so before proceeding with the rest of the development.

Univalence and computation. The scenarios above assume that univalent transport is *effective*: given a closed term of type $P\ A$, transport yields a closed term of type $P\ B$. However, in the Calculus of Inductive Constructions (CIC) or Martin-Löf Type Theory, univalence is expressed as an *axiom* [25]. The univalence axiom can be used in particular to establish what we hereby call the *Indiscernibility of Equivalents*,[1] formally that $A \simeq B$ implies that $P\ A \simeq P\ B$, 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 allows existing proof assistants such as Coq—and the vast amount of developments in these systems—to directly benefit from advances in this regard, while contributing to the general 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 [23] coined *univalent parametricity*. By supporting the justification of univalent parametricity per type constructors, we can, on a case-by-case basis, 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:

- We introduce *univalent parametricity* as a strengthening of parametricity to ensure preservation of equivalences (§ 3).
- We provide a logical relation for univalent parametricity defined over type constructors (§ 3.1). The principle of indiscernibility of equivalents for a type constructor amounts to the fundamental property of this logical relation (§ 3.2). We prove that each type constructor of the Calculus of Constructions with universes CC_ω is univalently parametric, identifying in each case the necessary assumptions (§ 3.3).

[1] Akin to the *indiscernibility of identicals a.k.a.* Leibniz's Law. To the best of our knowledge, the notion of *indiscernibility of equivalents* was introduced, in a different context, by the philosopher and logician Bacon [4].

- We also define univalent parametricity through a translation in the style of Bernardy *et al.* [7], which allows us to prove an abstraction theorem that entails that all terms of CC_ω are univalently parametric (§ 3.4).
- We extend univalent parametricity from CC_ω to CIC by dealing with inductive types (§ 4).
- The logical relation for univalent parametricity serves as the foundation for an ad-hoc realization of univalent parametricity in Coq with type classes [24], which is readily applicable to existing Coq developments, such as our introductory example (§ 5).
- We study the impact of the use of axioms in parts of proofs of univalent parametricity on the effectiveness of the induced univalent transport (§ 6). We show how to exploit the ad-hoc setting to maximize transport in specific situations through specialize type class instances. Finally we identify a useful fragment of CIC for which univalent transport is guaranteed to be effective.

The technical content of this work is fully formalized and proven in Coq (v8.7), including the translation and its properties, the type class framework and its instances, as well as the examples. The Coq source files are available as anonymous supplementary material.

Section 2 provides more precise background on type equivalence, univalence and parametricity in the context of dependent type theories. Section 7 discusses related work and Section 8 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 $f : A \to B$ is an *equivalence* iff there exists a function $g : B \to A$ 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 $A \simeq B$, iff there exists a function $f : A \to B$ 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 an 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 $(A = B) \simeq (A \simeq B)$. 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 $P : \mathit{Type} \to \mathit{Type}$, and any two types A and B such that $A \simeq B$, we have $P\ A \simeq P\ B$.*

Proof. Direct using univalence: $A \simeq B \implies A = B \implies P\ A = P\ B \implies P\ A \simeq P\ B$ □

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

Corollary 2.4 (Univalent Transport). *For any $P : \mathit{Type} \to \mathit{Type}$, 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 `List` $A \simeq$ `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_∀ : ∀ A B (P : A → Type) (Q : B → Type)
  (e : A ≃ B) (e' : ∀ b, P ↑( b) ≃ Q b), (∀ a, P a) ≃ (∀ b, Q b).

Definition equiv_functor_Σ : ∀ A B (P : A → Type) (Q : B → Type)
  (e : A ≃ B) (e' : ∀ a, P a ≃ Q ↑( a)), (Σ a, P a) ≃ (Σ a, Q a).
```

Such lemmas are sufficient to automatically derive an effective definition of the `head` function that operates on lists-with-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:

```
Lib Vector.t ≃ Lib (fun A n ⇒ {l: list A & length l = n})
```

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

```
∀ n A B (f : A → B) (l : {l:list A & length l = S n}),
  (head (map f ↑l) = f (head ↑l )) ≃  ↑(head ↑(map f l) = f ↑(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 automatization 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_functor_∀` and `equiv_functor_Σ`, 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 $A \to \textsc{Type}$ and $B \to \textsc{Type}$, 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 $\textsc{Type} \to \textsc{Type}$ to be able to relate type constructors à la F_ω as in our statement of Theorem 2.3. As a matter of fact, we also need the relation to be defined at $A \to \textsc{Type}$ 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 an heterogeneous relation, that would provide us with a powerful reasoning principle like the abstraction theorem of parametricity [23]. 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!"*.

2.4 Parametricity for dependent types

Reynolds originally formulated the relational interpretation of types to establish parametricity of System F [23]. Recently, Bernardy *et al.* [7] generalized the approach to pure type systems, including

$$
\begin{aligned}
[\![\textsc{Type}_i]\!]_p\ A\ B &\triangleq A \to B \to \textsc{Type}_i \\
[\![\Pi a : A.B]\!]_p\ f\ g &\triangleq \Pi(a : A)(a' : A')(e : [\![A]\!]_p\ a\ a').[\![B]\!]_p\ (f\ a)(g\ a') \\
[\![x]\!]_p &\triangleq x_r \\
[\![\lambda x : A.t]\!]_p &\triangleq \lambda(x : A)(x' : A')(x_r : [\![A]\!]_p\ x\ x').[\![t]\!]_p \\
[\![t\ u]\!]_p &\triangleq [\![t]\!]_p\ u\ u'\ [\![u]\!]_p \\
\\
[\![\cdot]\!]_p &\triangleq \cdot \\
[\![\Gamma, x : A]\!]_p &\triangleq [\![\Gamma]\!]_p, x : A, x' : A', x_r : [\![A]\!]_p\ x\ x'
\end{aligned}
$$

Fig. 1. Parametricity translation for CC_ω (from [7])

the Calculus of Constructions with universes CC_ω, 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_ω includes a hierarchy of universes $\textsc{Type}_i$, variables, applications, lambda expressions and dependent function types:

$$A, B, M, N ::= \textsc{Type}_i \mid x \mid M\ N \mid \lambda x : A.\ M \mid \Pi x : A.\ B$$

Its typing rules are standard, and hence omitted here—see [21] for a recent presentation.

Parametricity for CC_ω can be defined as a logical relation $[\![A]\!]_p$ for every type A. Specifically, $[\![A]\!]_p\ a_1\ a_2$ 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_ω allows the logical relation to be stated in CC_ω itself. Note that because terms and types live in the same world, $[\![-]\!]_p$ is defined for every term.

Figure 1 presents the definition of $[\![-]\!]_p$ for CC_ω, based on the work of [7]. For the universe $\textsc{Type}_i$, 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.

Armed with this translation, it is possible to prove an abstraction theorem à la Reynolds, saying that a well-typed term is related to itself (more precisely, to its duplicated-renamed self):

Theorem 2.5 (Abstraction theorem). *If* $\Gamma \vdash t : A$ *then* $[\![\Gamma]\!]_p \vdash [\![t]\!]_p : [\![A]\!]_p\ t\ t'$.

In particular, this means that the translation of a term $[\![t]\!]_p$ 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, $\vdash \textsc{Type}_i : \textsc{Type}_{i+1}$, this means that we have a kind of fixpoint property for the relation on $\textsc{Type}_i$:

$$\vdash [\![\textsc{Type}_i]\!]_p : [\![\textsc{Type}_{i+1}]\!]_p\ \textsc{Type}_i\ \textsc{Type}_i.$$

For parametricity, this property holds because

$$\lambda(A\ B : \textsc{Type}_i).\ \textsc{Type}_i : \textsc{Type}_i \to \textsc{Type}_i \to \textsc{Type}_{i+1}.$$

Note that this necessary fixpoint property is actually not trivial to satisfy in any variant of parametricity, as we will see in the next section.

The parametricity translation together with the abstraction theorem are powerful to derive free theorems (and proofs) [7]. However, they are insufficient to ensure preservation of equivalences. For example, for an arbitrary type constructor $P : \textsc{Type}_i \to \textsc{Type}_i$, the fundamental property tells us that the relation between two types A and B can be lifted to a relation between $P\ A$ and $P\ B$. However, even if we additionally assume that A and B are equivalent and that the relation between A and B is given by

$$\lambda(a : A)\ (b : B).\ a =\uparrow b,$$

we cannot freely conclude that $P\ A$ and $P\ B$ are themselves equivalent; indeed, we only know that $P\ A$ and $P\ B$ are in relation, without any additional constraint on this relation. Similarly, in our `Lib` example, we can show that `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 CC_ω; we defer scaling up to CIC to § 4.

We first define a *univalent logical relation* as a type-indexed logical relation on all the type constructors of CC_ω (§ 3.1). A term is *univalently parametric* if it is related to itself; in particular, we prove that univalently parametric constructors satisfy the Indiscernibility of Equivalents (§ 3.2). We discuss in § 3.3 the proofs that each type constructor is univalently parametric, paying attention to the potential use of axioms.

To prove that all well-typed terms of CC_ω are univalently parametric requires a definition of the relation that accommodates all terms of CC_ω, not just type constructors, including open terms. To do so, § 3.4 presents a translation for univalent parametricity in the style of Bernardy *et al.* [7]. For type constructors, the translation appeals to proof terms previously introduced in § 3.3.

Note that we present both descriptions of univalent parametricity because of their complementarity. The translation gives us an abstraction theorem and the general fundamental property for CC_ω. The univalent logical relation on type constructors allows us to relate terms of completely different types, such as inductively-defined and binary-encoded naturals. This is important because we want to be able to let programmers define their own equivalences. Additionally, the Coq formalization of the translation is based on a deep embedding, while the univalent logical relation is internalized directly through the type class system of Coq, hence bringing all the facilities of our approach to existing Coq developments (§ 5 and § 6).

3.1 Univalent logical relation

To strengthen parametricity to deal with equivalences, we need to strengthen the parametricity logical relation on the universe $\textsc{Type}_i$. Several intuitive solutions come to mind, which however are not satisfactory.

First, we could simply replace the heterogeneous relation demanded by parametricity to be type equivalence itself, *i.e.* $[\![\textsc{Type}_i]\!]_u\ A\ B \triangleq A \simeq B$. However, by doing so, the abstraction

$$
\begin{aligned}
A \approx B : \textsc{Type}_i \bowtie \textsc{Type}_i \;\triangleq\; & A \bowtie B \;\wedge\; A \simeq B \\
& \wedge\; \forall a : A, b : B, (a \approx b : A \bowtie B) \simeq (a = \uparrow b)
\end{aligned}
$$

$$
\begin{aligned}
P \approx Q : A \to \textsc{Type}_i \bowtie B \to \textsc{Type}_i \;\triangleq\; & A \bowtie B \\
& \wedge\; \forall a : A, \forall b : B, a \approx b : A \bowtie B \implies P\ a \approx Q\ b : \textsc{Type}_i \bowtie \textsc{Type}_i
\end{aligned}
$$

$$
\begin{aligned}
f \approx g : \Pi a : A.P\ a \bowtie \Pi b : B.Q\ b \;\triangleq\; & P \approx Q : A \to \textsc{Type}_i \bowtie B \to \textsc{Type}_i \\
& \wedge\; \forall a : A, \forall b : B, a \approx b : A \bowtie B \implies f\ a \approx g\ b : P\ a \bowtie Q\ b
\end{aligned}
$$

Fig. 2. Univalent relation for CC_ω

theorem fails on $\vdash \textsc{Type}_i : \textsc{Type}_{i+1}$. We would need to establish the fixpoint on the universe, *i.e.* $[\![\textsc{Type}_i]\!]_u : [\![\textsc{Type}_{i+1}]\!]_u\ \textsc{Type}_i\ \textsc{Type}_i$, but we have

$$[\![\textsc{Type}_i]\!]_u : \textsc{Type}_i \to \textsc{Type}_i \to \textsc{Type}_{i+1} \neq \textsc{Type}_i \simeq \textsc{Type}_i.$$

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

Another intuitive approach is to state that $[\![\textsc{Type}_i]\!]_u\ A\ B$ requires *both* an heterogeneous relation on A and B *and an equivalence* between A and B. While this goes in the right direction, it is insufficient because there is no connection between the two notions. This in particular implies that, when scaling up from CC_ω to CIC, the identity type—which defines the notion of equality—will not satisfy the fundamental property of the logical relation. 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 Figure 2 and noted $x \approx y : X \bowtie Y$, which relates two terms x and y of possibly different types X and Y, and is defined over all the type constructors of CC_ω.[2] We write simply $X \bowtie Y$ to specify that the univalent relation is defined between X and Y, *i.e.* $\cdot \approx \cdot : X \bowtie Y$ is defined.

At $\textsc{Type}_i$, the univalent relation $A \approx B : \textsc{Type}_i \bowtie \textsc{Type}_i$ requires both $A \bowtie B$ and $A \simeq B$, as well as a *coherence condition* between the heterogeneous relation and equality. This (crucial!) condition stipulates that the heterogeneous relation does coincide with propositional equality up to a transport using the equivalence, *i.e.*:

$$(a \approx b : A \bowtie B) \simeq (a = \uparrow b)$$

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 `equiv_functor_∀` do (§ 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

$$
\begin{aligned}
& \forall A\ B : \textsc{Type},\ \forall a\ a' : A,\ \forall\ b\ b' : B', \\
& \quad a \approx b : A \bowtie B \;\wedge\; a' \approx b' : A \bowtie B \implies a = a' \simeq b = b'
\end{aligned}
$$

which is possible only if we know that related inputs are *equal* up to transport.

[2]Note that the domain of the logical relation is expressible in CIC.

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 relation 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 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 $\text{Univ}(x)$, iff $x \approx x : X \bowtie X$.

Using the univalent relation presented above, we cannot establish its fundamental property (namely, that all well-typed CC_ω terms are univalently parametric); we will do so in § 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. *Univ(P), then for all $A, B : \text{Type}_i$, $A \simeq B \implies P\ A \simeq P\ B$.*

Proof. If we unfold the definition, $\text{Univ}(P)\ A\ B$ means that

$$A \approx B : \text{Type}_i \bowtie \text{Type}_i \implies P\ A \approx P\ B : \text{Type}_i \bowtie \text{Type}_i$$

Because we know that $A \simeq B$, we can build the canonical heterogeneous relation

$$\lambda(a : A)(b : B).a = \uparrow b$$

which trivially satisfies the coherence condition, so $A \approx B : \text{Type}_i \bowtie \text{Type}_i$. Therefore, $P\ A \approx P\ B : \text{Type}_i \bowtie \text{Type}_i$, which in particular means that $P\ A \simeq P\ B$. □

Note that in the proof, we use the canonical relation $\lambda(a : A)(b : B).a = \uparrow b$, which uses both equality and univalent transport, to get a term $\text{canon}(e) : A \approx B : \text{Type}_i \bowtie \text{Type}_i$ from $e : A \simeq B$. 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 were to eagerly impose the use of this relation, we would be back in transport hell (§ 2.3).

3.3 Type constructors are univalently parametric

We now prove that the universe Type_i and the dependent function type Π 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. *Univ(Type_i) is inhabited.*

Proof. First, we need to define a relation between Type_i and Type_i. By a fixpoint argument, it has to be $\text{Type}_i \bowtie \text{Type}_i$. We also need to provide an equivalence $\text{Type}_i \simeq \text{Type}_i$; we simply take

the identity equivalence $\mathrm{id}_{\textsc{Type}_i}$. Finally, we need to prove that the relation is coherent with equality, that is, we need to exhibit a term $\mathsf{univ}_{\textsc{Type}_i}$ such that:

$$\mathsf{univ}_{\textsc{Type}_i} : \Pi A\ B.\ (A \approx B : \textsc{Type}_i \bowtie \textsc{Type}_i) \simeq (A = B)$$

For the function from $A = B$ to $A \approx B : \textsc{Type}_i \bowtie \textsc{Type}_i$, by induction on equality, it is sufficient to provide the canonical inhabitant $\mathsf{canon}(\mathrm{id}_A) : A \approx A : \textsc{Type}_i \bowtie \textsc{Type}_i$ associated 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 of the section and the retraction of the equivalence postulated by the univalence axiom, together with lemmas about decomposition of equality over $\textsc{Type}_i \bowtie \textsc{Type}_i$ and commutation of transports; the interested reader can consult the Coq development. As it uses the univalence axiom in its section and retraction, this equivalence is not entirely effective. □

3.3.2 Prop. In our definition, Prop is treated in the same way as $\textsc{Type}_i$ because $\textsc{Prop} : \textsc{Type}_i$ 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 $\mathsf{Univ}(\textsc{Type}_i)$ 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:

$$(P = Q) \simeq (P \iff Q).$$

This axiom can not be deduced from univalence alone, one would need proof irrelevance for Prop as well. 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 that Prop is proof irrelevant, $\textsc{Prop} \bowtie \textsc{Prop}$ boils down to

$$A \bowtie B \ \wedge\ A \iff B \ \wedge\ \forall a : A, b : B,\ \mathtt{IsContr}(a \approx b : A \bowtie B).$$

where `IsContr` A says that A is contractible, *i.e.* has a unique inhabitant. This is because for all a and b, the type $(a = \uparrow b)$ 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 § 7).

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

$$f = g \to \Pi(x : A).f\ x = g\ x$$

is an equivalence. This property is a consequence of univalence [25].

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

Proof. $\mathsf{Univ}(\Pi)\ A\ B\ P\ Q$ unfolds to

$$\begin{aligned} &A \approx B : \textsc{Type}_i \bowtie \textsc{Type}_i \to P \approx Q : A \to \textsc{Type}_i \bowtie B \to \textsc{Type}_i \\ &\to \Pi(a : A).\ P\ a \approx \Pi(b : B).\ Q\ b : \textsc{Type}_i \bowtie \textsc{Type}_i \end{aligned}$$

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

Next, we need to show that $\Pi(a : A).\ P\ a \simeq \Pi(b : B).\ Q\ b$ knowing that $A \simeq B$ and $\Pi(a : A)\ (b : B).a \approx b : A \bowtie B \to P\ a \simeq Q\ b$. Using the equivalence between $a \approx b : A \bowtie B$ and $a = \uparrow b$, this boils down to $\Pi(a : A).\ P\ a \simeq Q\ (\uparrow a)$.

At this point we can apply a standard result of HoTT [25], namely `equiv_functor_∀` in the Coq HoTT library [5], which was already introduced in § 2.3. This lemma requires functional extensionality in the proof that the two transport functions form an equivalence.[3]

We note the resulting term Equiv_Π, with:

$$\begin{aligned}\text{Equiv}_\Pi : \Pi A\ B\ P\ Q.\ A \simeq B \to \\ (\Pi(a : A)\ (b : B).a \approx b : A \bowtie B \to P\ a \simeq Q\ b) \to \\ \Pi(a : A).\ P\ a \simeq \Pi(b : B).\ Q\ b\end{aligned}$$

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

$$\text{univ}_\Pi : \Pi f\ g.(f \approx g : \Pi a : A.\ P\ a \bowtie \Pi b : B.\ Q\ b) \simeq (f = \uparrow g)$$

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 (§ 2.3). This is done by repeated use of commutativity lemmas of transport of equality over functions. Again, the interested reader can consult the Coq development. □

3.4 Univalent parametricity translation

Proving the general fundamental theorem of univalent parametricity requires an induction on the whole syntax of CC_ω, 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 § 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_ω but rather CIC_u, which is CIC plus the univalence axiom. We note $\Gamma \vdash_u t : T$ 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 $[\![A]\!]_u^{eq}$ and a witness $[\![A]\!]_u^{coh}$ that the relation is coherent with equality.

As explained in § 3.1, for Type_i, following Figure 2 but switching to the type theoretical notation, we want to set[4]:

$$\begin{aligned}[\![\text{Type}_i]\!]_u\ A\ B \triangleq\ & \Sigma(R : A \to B \to \text{Type}_i)(e : A \simeq B). \\ & \Pi a\ b.(R\ a\ b) \simeq (a = \uparrow b).\end{aligned}$$

That is, the translation of a type (when seen as a term) needs to include the parametricity relation plus the fact that there is an equivalence, and that the relation is coherent with equality. It is thus a dependent 3-tuple,[5] as explicit in Figure 3.

We therefore need to distinguish between the translation of a type T occurring in a *term position* (*i.e.* left of the ":"), translated as $[T]_u$ and the translation of a type T occurring in a *type position*

[3]The definition of the inverse function requires using the retraction, and the proof that it forms a proper equivalence requires the adjunction condition (§ 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.
[4]The notation $\Sigma a : A.\ B$ is a dependent pair, defined in CIC as an inductive type.
[5]We introduce syntactic sugar $t = (a; b; c)$ with accessors $t.1$ $t.2$ and $t.3$ for nested pairs to ease the reading.

$$
\begin{aligned}
[\textsc{Type}_i]_u &\triangleq (\lambda\ (A\ B : \textsc{Type}_i),\ \Sigma(R : A \to B \to \textsc{Type}_i)(e : A \simeq B).\\
&\qquad \Pi ab.(R\ a\ b) \simeq (a =\uparrow b); \mathrm{id}_{\textsc{Type}_i}; \mathsf{univ}_{\textsc{Type}_i})\\
[\Pi a : A.B]_u &\triangleq (\lambda\ (f\ g : \Pi a : A.B),\ \Pi(a : A)(a' : A')(a_r : [\![A]\!]_u\ a\ a').\\
&\qquad [\![B]\!]_u\ (f\ a)(g\ a'); \mathrm{Equiv}_\Pi\ [\![A]\!]_u^{eq}\ [\![B]\!]_u^{eq}; \mathsf{univ}_\Pi)\\
[x]_u &\triangleq x_r\\
[\lambda x : A.t]_u &\triangleq \lambda(x : A)(x' : A')(x_r : [\![A]\!]_u\ x\ x').[t]_u\\
[t\ u]_u &\triangleq [t]_u\ u\ u'\ [u]_u
\end{aligned}
$$

$$
[\![A]\!]_u \triangleq [A]_u.1 \quad [\![A]\!]_u^{eq} \triangleq [A]_u.2 \quad [\![A]\!]_u^{coh} \triangleq [A]_u.3
$$

$$
\begin{aligned}
[\![\cdot]\!]_u &\triangleq \cdot\\
[\![\Gamma, x : A]\!]_u &\triangleq [\![\Gamma]\!]_u, x : A,\ x' : A',\ x_r : [\![A]\!]_u\ x\ x'
\end{aligned}
$$

Fig. 3. Univalent parametricity translation for CC_ω

(*i.e.* right of the ":"), translated as $[\![T]\!]_u$.[6] The fundamental property on $\textsc{Type}_i$ enforces the definition of the relation, equivalence and coherence on a type T to be deduced from $[T]_u$ respectively as

$$
[\![A]\!]_u \triangleq [A]_u.1 \quad [\![A]\!]_u^{eq} \triangleq [A]_u.2 \quad [\![A]\!]_u^{coh} \triangleq [A]_u.3
$$

The 3-tuples for $\textsc{Type}_i$ and dependent function type are precisely given by the fact that they are in the diagonal of the univalent relation, as proved in § 3.3. In particular, the terms $\mathsf{univ}_{\textsc{Type}_i}$ and univ_Π used in the translation have been described in Proposition 3.3 and Proposition 3.5. Note that 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* $[\![\Gamma]\!]_u \vdash_u [t]_u : [\![A]\!]_u\ t\ t'$.

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_ω is univalently parametric.

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

Proof. For a closed term, we have $[\![A]\!]_u \equiv A \bowtie A$ and $a = a'$, so by the abstraction theorem, $[a]_u : a \approx a : A \bowtie A$. □

[6] The possibility to distinguish the translation of a type on the left and right-hand side of a judgment has already been noticed for other translations that add extra information to types by Boulier *et al.* [9]. For instance, to prove the independence of univalence with CIC, they use a translation that associates a Boolean to any type, *e.g.* $[\textsc{Type}_i] = (\textsc{Type}_i \times \mathbb{B}, \mathsf{true})$. Then a type on the left-hand side is translated as a 2-tuple and $[\![A]\!] = [A].1$. This possibility to add additional information in the translation of a type comes from the fact that types in CIC can only be "observed" through inhabitance, that is, in a type position; therefore, the translation in term positions may collect additional information.

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 (§ 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 N, *i.e.* $\mathbb{N} \approx \mathsf{N} : \textsc{Type} \bowtie \textsc{Type}$.

4 UNIVALENT PARAMETRICITY AND INDUCTIVE TYPES

CIC is an extension of CC_ω that allows for the definition of inductive types in the theory. An inductive type is defined as a new type constructor, together with associated constructors and an elimination principle.[7] For instance, the inductive type of lists is[8]

```
Inductive list (A : Type) : Type :=
  nil : list A
| cons : A → list A → list A
```

where `nil` and `cons` are the constructors of the inductive type. The associated eliminator is

```
list_rect : ∀ (A : Type) (P : list A → Type), P nil →  (∀ (a : A) (l : list A), P l → P (a :: l))
          → ∀ l : list A, P l.
```

To prove that a given inductive type I (with constructors I_c_i and elimination I_rect) is univalent—and thus being able to extend the abstraction theorem of CC_ω—one needs to prove $\mathrm{Univ}(I)$, $\mathrm{Univ}(I\_c_i)$ and $\mathrm{Univ}(I\_rect)$.

We proceed step-by-step, considering first dependent pairs (§ 4.1), then records (§ 4.2), parameterized recursive inductive families (§ 4.3), and finally indexed inductive types (§ 4.4). Proofs of univalent parametricity for inductives do not require using axioms, however they potentially propagate the axioms used in the equivalence proofs of their parameters and indices. We discuss practical strategies to achieve effectiveness in § 6.

4.1 Dependent pairs

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

```
Inductive sigT (A : Type) (B : A → Type) : Type :=
    existT : ∀ x : A, B x → {x : A & B x}.
```

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

```
sigT_rect : ∀ (A : Type) (P : A → Type) (P₀ : sigT A P → Type),
    (∀ (x : A) (p : P x), P₀ (x; p)) → ∀ s : sigT A P, P₀ 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 [20].

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 4.1. *$\mathrm{Univ}(\Sigma)$ is inhabited.*

[7] There is an equivalent presentation of inductive types with pattern matching instead of eliminators. In Coq, eliminators are automatically inferred and defined using pattern matching.

[8] In this section, to ease the reading, we navigate between the syntax of CIC and the one of Coq when appropriate.

$$p \approx q : \Sigma a : A.P\ a \bowtie \Sigma b : B.Q\ b \triangleq P \approx Q : A \to \text{Type} \bowtie B \to \text{Type}$$
$$\wedge\ p.1 \approx q.1 : A \bowtie B\ \wedge\ p.2 \approx q.2 : P\ p.1 \bowtie Q\ q.1$$

Fig. 4. Univalent relation for dependent pairs

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

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

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 `existT` and the eliminator `sigT_rect` are univalently parametric are direct by induction on the structure of a dependent pair type.

4.2 Dependent records

Let us go back to the example of the `Lib` record type

```
Record Lib (C : Type → ℕ → Type) :=
 { head : ∀ {A : Type} {n : ℕ}, C 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)}.
```

Like all record types, `Lib` can be formulated in terms of nested dependent sums. This means that, for any `C : Type → ℕ → Type`, `Lib C` is equivalent to

```
Lib' C := Σ (hd : ∀ A n. C A (S n) → A).
          Σ (map : ∀ A B (f:A → B) n, C A n → C B n).
            ∀ n A B (f : A → B) (v : C A (S n)), hd (map f v) = f (hd v).
```

The fact that `Lib'` is univalent directly follows from the abstraction theorem of CC_ω extended with dependent sums. To conclude that `Lib` is univalent, we use the fact that a type family equivalent to a univalent type family is itself univalent.

Proposition 4.2. *Let $X : \text{Type}_i$, $A\ B : X \to \text{Type}_j$, and $x\ y : X$ such that $X \approx X$, $x \approx y$, and $A\ x \approx A\ y$. If $\forall x, B\ x \simeq A\ x$ then $B\ x \approx B\ y$.*

Proof. Follows from the fact that, for all types $A\ B\ C$, we have $A \approx B\ \wedge\ B \simeq C \implies A \approx C$. □

This approach to establish the univalence of a record type via its encoding with dependent sums can be extended to any record type. We do not present here the generalized version, but we have automatized this principle in our Coq framework as a tactic, by reusing an idea used in the HoTT library [5] that allows automated inference of type equivalence for records with their nested pair types formulation. This tactic can be used to automatically prove that a given record type is univalently parametric (provided its fields are).

4.3 Parameterized recursive inductive families

To establish the univalent parametricity of a *parameterless* recursive inductive type I, such as natural numbers with zero and successor, we can simply use the canonical structure over the identity equivalence, with equality as univalent relation and trivial coherence: canon($Equiv\_id\ I : I \simeq I$) : Univ(I).

However, whenever an inductive type has parameters, the situation is more complex.[9] Let us develop the case of lists. First, we need to show that

$$\Pi(A\ B : \text{Type}_i).\ (A \simeq B) \to (\texttt{list}\ A \simeq \texttt{list}\ B).$$

The two transport functions of the equivalence $\texttt{list}\ A \simeq \texttt{list}\ B$ can be defined by induction on the structure of the list (*i.e.* using the eliminator `list_rect`). They both simply correspond to the usual map operation on lists. The proof of the section and retraction are also direct by induction on the structure of the list, and transporting along the section and retraction of $A \simeq B$.

The univalent relation on lists is given directly by parametricity. Indeed, following the work of Bernardy *et al.* on the inductive-style translation [7], the inductive type corresponding to the lifting of a relation between `A` and `B` to a relation between `list A` and `list B` is given by:

```
Inductive UR_list A B (R : A → B → Type) : list A → list B → Type :=
 UR_list_nil : UR_list R nil nil
| UR_list_cons : ∀ a b l l', (R a b) → (UR_list R l l') → UR_list R (a::l) (b::l').
```

This definition captures the fact that two lists are related if they are of the same length and pointwise-related. Then, the univalent relation is given by

$$l \approx l' : \texttt{list}\ A \bowtie \texttt{list}\ B \triangleq \texttt{UR\_list}\ \texttt{A}\ \texttt{B}\ (\texttt{A} \bowtie \texttt{B})$$

Similarly to dependent pairs, the proof that the relation is coherent with equality relies on the following decomposition of equality between lists:

$$\Pi\ A\ B\ (e : A \simeq B)\ l\ l'.\ (\texttt{UR\_list}\ A\ B\ (\lambda\ a\ b.a = \uparrow b)\ l\ l') \simeq (l = \uparrow l').$$

Indeed, using this lemma, the coherence of the univalent relation with equality is easy to infer:

$$(l \approx l') \equiv \texttt{UR\_list}\ \texttt{A}\ \texttt{B}\ (\texttt{A} \bowtie \texttt{B}) \simeq (\texttt{UR\_list}\ \texttt{A}\ \texttt{B}\ (\lambda\ a\ b.a = \uparrow b)\ l\ l') \simeq (l = \uparrow l')$$

Note that it is always valid to decompose equality on inductive types. This is because values of an inductive type can only be observed by analyzing which constructor was used to build the value. This fact is explicitly captured by the elimination principle of an inductive type. On the contrary, for dependent products, the fact that functions can only be observed through application to a term is implicit in CIC, *i.e.* there is no corresponding elimination principle in the theory (hence functional extensionality is an axiom).

The proofs that the constructors `nil` and `cons` are univalent are direct by definition of `UR_List`. Likewise, the proof that the eliminator `list_rect` is univalent is direct by induction on `UR_List`.

Generalization. It is possible to generalize the above result developed for lists to any parameterized inductive family, although a general proof is outside the scope of this paper. As illustrated above, the univalent relation for parameterized inductive families is given by parametricity, and the proof that related inputs give rise to equivalent types proceeds by a direct induction on the structure of the type. The main difficulty is to generalize the proof of the coherence of the relation with equality. Indeed, this involves fairly technical reasoning on equality and injectivity of constructors.

Fortunately, in practice in our Coq framework, a general construction is not required to handle each new inductive type I, because a witness of the fact that a given inductive I is univalent can be defined specifically as a typeclass instance. We also provide a tactic to automatically generate this

[9] In this work the distinction between *parameters* and *indices* for inductive types is important. A parameter is merely indicative that the type behaves *uniformly* with respect to the supplied argument. For instance `A` in `list A` is a parameter. Thus the choice of `A` only affects the type of elements inside the list, not its shape. In particular, by knowing `A` for a given list, we cannot infer which constructor was used to construct that list. On the other hand, by knowing the value of an index, one can infer which constructor(s) may or may not have been used to create the value. For instance, a value of type `Vect A 0` is necessarily the empty vector.

proof on any parameterized datatypes (up to a fixed number of constructors), depending on the univalent parametricity of its parameters.

4.4 Indexed inductive families

CIC allows defining inductive types that are not only parameterized, but also indexed, like length-indexed vectors `Vector` A n. Another mainstream example is Generalized Algebraic Data Types [22] (GADTs) illustrated here with the typical application to modeling typed expressions:

```
Inductive Expr : Type → Type :=
| I : ℕ → Expr ℕ
| B : 𝔹 → Expr 𝔹
| Ad : Expr ℕ → Expr ℕ → Expr ℕ
| Eq : Expr ℕ → Expr ℕ → Expr 𝔹.
```

Observe that the return types of constructors instantiate the inductive family at specific type indices, instead of uniform type parameters as is the case for *e.g.* the parameterized list inductive type. This specificity of constructors is exactly what makes GADTs interesting for certain applications; but this is precisely why their univalent parametricity is ineffective!

Indeed, consider an equivalence between natural numbers $\mathbb{N}$ and binary natural numbers `N`. Univalence of the `Expr` GADT means that `Expr` $\mathbb{N}$ is equivalent to `Expr N`. However, there is no constructor for `Expr` that can produce a value of type `Expr N`. So the only way to obtain such a term is by using an *equality* between $\mathbb{N}$ and `N`, that is, using the univalence axiom.[10]

The challenge is that univalence for indexed inductive families relies on the coherence condition. To better understand this point, let us study the prototypical case of identity types.

Identity types. In Coq, the identity type (or equality type) is defined as an indexed inductive family with a single constructor `eq_refl`:

```
Inductive eq (A : Type) (x : A) : A → Type := eq_refl : x = x.
```

The elimination principle `eq_rect`, known as *path induction* in HoTT terminology, is:

```
eq_rect : ∀ (A : Type) (x : A) (P : ∀ a : A, x = a → Type), P x eq_refl → ∀ (y : A) (e : x = y), P y e
```

Proposition 4.3. *Univ*(`eq`) *is inhabited.*

Proof. Univ(eq) unfolds to

$$\Pi\ (A\ B : \textsc{Type})\ (a\ a' : A)\ (b\ b' : B)\ (eAB : A \approx B)\ (e : a \approx b)\ (e' : a' \approx b'), (a =_A a') \bowtie (b =_B b')$$

To prove that $(a = a') \simeq (b = b')$, it is first necessary to transform $a \approx b$ and $a' \approx b'$ using the fact that $A \bowtie B$ and hence that the relation is coherent with equality. After rewriting, the equivalence to establish is

$$(a =_A a') \simeq (\uparrow a =_B \uparrow a')$$

This equivalence is similar to a standard result of HoTT [25], namely `equiv_functor_eq` in the Coq HoTT library [5], which was already introduced in § 2.3.

The univalent relation for identity types is defined using the inductive type that is obtained by applying parametricity to the identity type:

[10] It is however impossible to *prove* that no term of type `Expr N` can be constructed without univalence, because the univalence axiom is compatible with CIC.

```
Inductive UR_eq (A₁ A₂ : Type) (A_R : A₁ → A₂ → Type) (x₁ : A₁) (x₂ : A₂) (x_R : A_R x₁ x₂) :
  ∀ (y₁ : A₁) (y₂ : A₂), A_R y₁ y₂ → x₁ = y₁ → x₂ = y₂ → Type :=
  UR_eq_refl : UR_eq A₁ A₂ A_R x₁ x₂ x_R x₁ x₂ x_R eq_refl eq_refl.
```

The univalent relation is just a specialization of `UR_eq` where `A_R` is given by $\approx$ on A and B:

$$e_1 \approx e_2 : a =_A a' \bowtie b =_B b' \triangleq \texttt{UR\_eq}\ A\ B\ \approx\ a\ b\ e\ a'\ b'\ e'\ e_1\ e_2$$

Finally, proving that the relation is coherent with equality amounts to show that

$$\Pi(e_1\ e_2 : a = a').\ (e = e') \simeq \texttt{UR\_eq}\ A\ B\ (A \bowtie B)\ a\ b\ e\ a'\ b'\ e'\ e_1\ \uparrow e_2.$$

This can be done be first showing the following equivalence[11]

$$\texttt{UR\_eq}\ A\ B\ P\ x\ y\ H\ x'\ y'\ H'\ X\ Y \simeq (Y\ \#\ (X\ \#\ H) = H')$$

which means that the naturality square between H and H' commutes.

The proofs that `eq_refl` and `eq_rect` are univalent are direct by `UR_eq_refl` and elimination of `UR_eq`. □

To deal with other indexed inductive types, one can follow a similar approach. Alternatively, it is possible to exploit the correspondence between an indexed inductive family and a subset of parameterized inductive family, established by [13], to prove the univalence of an indexed inductive family. In this correspondence, the property of the subset type is obtained from the identity type.

For instance, for vectors:

$$\texttt{Vector}\ A\ n \simeq \Sigma\ l : \texttt{list}\ A.\ \texttt{length}\ l = n$$

The `length` function computes the length of a list, as follows:

```
Definition length {A} (l: list A) : ℕ := list_rect A (fun _ ⇒ ℕ) 0 (fun _ l n ⇒ S n) l
```

where one can observe that the semantics of the index in the different constructors of vectors is captured in the use of the recursion principle `list_rect`. By the abstraction theorem, $\Sigma\ l : \texttt{list}\ A.\ \texttt{length}\ l = n$ is univalent, and thus by Proposition 4.2, so is `Vect` $A\ n$.

5 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 (§ 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 (§ 5.1), and then describe the instances for some type constructors (§ 5.2).

5.1 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:[12]

```
Class IsEquiv (A B : Type) (f:A → B) := {
  e_inv : B → A ;
  e_sect : ∀ x, (e_inv ∘ f) x = x;
  e_retr : ∀ y, (f ∘ e_inv) y = y;
```

[11] The notation $e\ \#\ t$, with $e : x = y$ and $t : P\ x$ when P is clear from the context, denotes the transport of the term e through the equality proof e (hence $e\ \#\ t : P\ y$).

[12] Adapted from: http://hott.github.io/HoTT/coqdoc-html/HoTT.Overture.html.

```
  e_adj  : ∀ x, e_retr (f x) = ap f (e_sect x) }.
```

The properties `e_sect` and `e_retr` express that `e_inv` is both the left and right inverse of `f`, respectively. The property `e_adj` is a compatibility condition between the proofs. It ensures that the equivalence is uniquely determined by the function `f`.

While `IsEquiv` characterizes a particular function f as being an equivalence, we say that two types A and B are equivalent, noted $A \simeq B$, iff there exists such a function f.

```
Class Equiv A B := { e_fun :> A → B ; e_isequiv : IsEquiv e_fun }.
Notation "A ≃ B" := (Equiv A B).
```

`Equiv` is here defined as a type class to allow automatic inference of equivalences. This way, we can define automatic transport as

```
Definition univalent_transport {A B : Type} {e: A ≃ B} : A → B := e_fun e.
Notation "↑" := univalent_transport.
```

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

To formalize univalent relations, we define a hierarchy of classes, starting from `UR` for univalent relations (arbitrary heterogeneous relations), refined by `UR_Coh`, which additionally requires the proof of coherence between a univalent relation and equality.

```
Class UR A B := { ur : A → B → Type }.
Notation "x ≈ y" := (ur x y) (at level 20).

Class UR_Coh A B (e : Equiv A B) (H: UR A B) := { ur_coh : ∀ (a a' : A), Equiv (a = a') (a ≈ ↑( a')) }.
```

As presented in Figure 3, two types are related by the univalent parametricity relation if they are equivalent and there is a coherent univalent relation between them. This is captured by the typeclass `UR_Type`.

```
Class UR_Type A B := {
  Ur :> UR A B;
  equiv :> A ≃ B;
  Ur_Coh:> UR_Coh A B equiv Ur;
  Ur_Can_A :> Canonical_eq A;
  Ur_Can_B :> Canonical_eq B }.
Infix "⋈" := UR_Type.
```

(The last two attributes are part of the Coq framework in order to better support extensibility and effectiveness, as will be described in § 6.2.)

5.2 Univalent type constructors

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

For the universe Type_i, we define:

```
Instance UR_Type_def@{i j} : UR@{j j j} Type@{i} Type@{i} := {| ur := UR_Type@{i i i} |}.
```

This is where our fixpoint construction appears: the relation at Type_i is defined to be `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 `UR_Type_def` instance will be used implicitly everywhere we use the notation `X ≈ Y`, when `X` and `Y` are types themselves.[13]

For dependent function types, we set:

```
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 x) (∀ y, B' y) :=
  {| ur := fun f g ⇒ ∀ x y (H: x ≈ y), f x ≈ g y |}.
```

The univalent parametricity relation on dependent function types expects relations on the domain and codomain types, the latter being parameterized by the former through its argument `(H : x ≈ y)`. The definition is the standard heterogeneous extensionality principle on dependent function types.

Interestingly, the `Equiv` instance derived from this definition for dependent function types has the following type:

```
Instance Equiv_∀ : ∀ (A A' : Type) (e_A : A ≈ A') (B : A → Type)
  (B' : A' → Type) (e_B : B ≈ B'),  (∀ x : A, B x) ≃ (∀ x : A', B' x).
```

While the conclusion is an equivalence, the assumptions `eA` and `eB` are about univalent relations for `A`, `A'` and `B` and `B'`. The first one is implicitly resolved as the `UR_Type_def` defined above, and the second one as a combination of `UR_Forall` and `UR_Type_def`. With these stronger assumptions, and because ≈ is heterogeneous, we can prove the equivalence *without introducing transports*, and hence avoid the transport hell (§ 2.3). This is key to make the type class instance proof search tractable: it is basically structurally recursive on the type indices. We can then show that the dependent function type seen as a binary type constructor is related to itself using the univalent relation and equivalence constructed above:

```
Definition FP_∀ : (fun A B ⇒ (∀ x:A , B x)) ≈ (fun A' B' ⇒ (∀ x:A', B' x)).
```

To instrument the type class instance proof search, we add proof search hints for each fundamental property.

We proceed similarly for other constructors, *i.e.* dependent pairs, the identity type, natural numbers and booleans with the canonical univalent relation, where we additionally prove the fundamental property for the eliminators; *i.e.* we have many fundamental property lemmas such as:

```
Definition FP_Σ : @sigT ≈ @sigT.
```

Having spelled out the basics of the Coq framework for univalent parametricity, we can now turn to the practical issue of effective transport.

6 EFFECTIVENESS OF UNIVALENT TRANSPORT

Univalence is not constructively expressible in CIC, hence univalent transport is not generally effective. Univalent parametricity allows us to address each constructor of the theory in turn, and to avoid axioms altogether or push them "as far back as possible". However, because some proofs of univalent parametricity rely on axioms, transported functions that exercise these proofs will not be effective. Understanding when these proofs can be "exercised" is therefore crucial to then devise techniques for maximizing effectiveness of transport.

[13]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 `@{j j j}` above), although some annotations appear in the Coq source files in order to explicitly validate our assumptions about universes.

6.1 Axioms in Univalent Parametricity Proofs and Effectiveness of Transport

The proofs of univalent parametricity we have developed make use of two axioms:

(1) The univalence axiom is used to show the coherence condition of univalent parametricity for the universe (§ 3.3.1). This is to be expected and unavoidable, as this condition for the universe exactly states that type equivalence coincides with equality.
(2) The functional extensionality axiom is used to show that the transport functions of the equivalence for the dependent product form an equivalence (§ 3.3.3).

Additionally, as shown in § 4, the effectiveness of univalent transport for inductive types depends on the type of parameters and indices. In particular, proving univalent parametricity of indexed families requires using the coherence condition.

To see how this relates to practice, consider the case of functions (2). Functional extensionality is only used in the proof that the transport functions form an equivalence. In particular, this means that the transport functions themselves *are* effective. Therefore, when transporting a first-order function, the resulting function is effective.

For the axiom to interfere with effectiveness, we need to consider a *higher-order* function, *i.e.* that takes another function as argument. Consider for instance the conversion of a higher-order dependent function g operating on a function over natural numbers

```
g : ∀ (f: ℕ → ℕ), Vector ℕ (f 0)
```

to one operating on a function over binary natural numbers

```
g' : ∀ (f: N → N), Vector N (f ↑(0)) : ↑g.
```

We transport g to g' along the equivalence between the two higher-order types above. Such a transport uses, *in a computationally-relevant position*, the fact that the function argument f can be transported along the equivalence between $\mathbb{N} \to \mathbb{N}$ and N → N. Consequently, the use of functional extensionality in the equivalence proof shimes in, and g' is not effective. (Specifically, g' pattern matches on an equality between natural numbers that contains the functional extensionality axiom.)

Fortunately, there are different ways to circumvent this problem, by exploiting the fact that univalent parametricity is defined in an *ad hoc* manner, and hence specializable through specific type class instances. This section shows how we can further specialize proofs of univalent parametricity in situations where using axioms can be avoided. Sometimes we can ignore the fact that an equality proof might be axiomatic by automatically crafting a new one that is axiom-free (§ 6.2), or we can avoid transporting type families with (potentially axiomatic) proofs of equality in some specific cases (§ 6.3 and § 6.4). Finally, we identify a syntactic fragment of CIC for which effective transport is *guaranteed* (§ 6.5). This fragment is practically relevant as it corresponds to standard cases of certified programming, *i.e.* it contains System F_ω and indexed inductive types with a decidable equality.

6.2 Canonical Equality for Types with Decidable Equality

Any proof of equality between two natural numbers can be turned into a canonical, axiom-free proof using decidability of equality on natural numbers. In general, decidable equality on a type A can be expressed in type theory as

```
Definition DecEq (A : Type) := ∀ x y : A, (x = y) + ¬(x = y).
```

Hedberg's theorem [25] implies that if A has decidable equality, then A satisfies Uniqueness of Identity Proofs (UIP): any two proofs of the same equality between elements of A are equal. Hedberg's theorem relies on the construction of a canonical equality to which every other is shown equal. Specifically, when A has a decidable equality, it is possible to define a function

```
Definition Canonical_eq_decidable A (Hdec : DecEq A) : ∀ x y : A, x = y → x = y :=
 fun x y e ⇒ match Hdec x y with
        | inl e0 ⇒ e0
        | inr n ⇒ match (n e) with end end.
```

This function produces an equality between two terms `x` and `y` of type `A` by using the decision procedure `Hdec`, independently of the equality `e`. In the first branch, when `x` and `y` are equal, it returns the canonical proof produced by `Hdec`, instead of propagating the input (possibly-axiomatic) proof `e`. And in case the decision procedure returns an inequality proof (of type `x=y → False`), the function uses `e` to establish the contradiction. In summary, the function transforms any equality into a canonical equality by using the input equality *only in cases that are not possible*.

We can take advantage of this insight to ensure effective transport on indices of types with decidable equality. The general idea is to extend the relation on types $A \bowtie B$ to also include two functions `∀ x y : A , x = y → x = y` and `∀ x y : B , x = y → x = y`. For types with decidable equality, these functions can exploit the technique presented above, and for others, these are just the identity. However, care must be taken: we cannot add arbitrary new computational content to the relation; we have to require that these functions preserve reflexivity. This is specified in the following class:

```
Class Canonical_eq (A:Type) :=
 { can_eq : ∀ (x y : A), x = y → x = y ;
   can_eq_refl : ∀ x, can_eq x x eq_refl = eq_refl }.
```

which is used for the last two attributes of the `UR_Type` class given in § 5.1.

There are two canonical instances of `Canonical_eq`, the one that is defined on types with decidable equality, and exploits the technique above, and the default one, which is given by the identity function (and proof by reflexivity).

Using this extra information, it is possible to improve the definition of univalent parametricity by always working with canonical equalities. This way, equivalences for inductive types whose indices are of types with decidable equality—like length-indexed vectors and many common examples—never get stuck on rewriting of indices.

6.3 Canonically Transportable Predicates

As mentioned in the introduction of this section, for some predicates, it is not necessary to pattern match on equality to implement transport.

The simplest example is when the predicate does not actually depend on the value, in which case `P x ≃ P y` can be implemented by the identity equivalence because `P x` is convertible to `P y`, independently of what `x` and `y` are. It is also the case when the predicate is defined on a type with a decidable equality, so we can instead pattern match on the canonical equality (§ 6.2).

To take advantage of this situation whenever possible, we introduce the notion of *transportable* predicates.

```
Class Transportable {A} (P : A → Type) := {
  transportable :> ∀ x y, x = y → P x ≃ P y;
  transportable_refl : ∀ x, transportable x x eq_refl = Equiv_id (P x) }.
```

Note that as for `Canonical_eq`, we need to require that `transportable` behaves like the standard transport of equality by sending reflexivity to the identity equivalence.

For instance, the instance for non-dependent functions is defined as

```
Instance Transportable_cst A B : Transportable (fun _ : A ⇒ B) := {|
  transportable := fun (x y : A) _ ⇒ Equiv_id B;
```

```
  transportable_refl := fun x : A ⇒ eq_refl |}.
```

To propagate the information that every predicate (a.k.a. type family) comes with its instance of `Transportable`, we specialize the definition of `UR (A → Type) (A' → Type)`:

```
Class URForall_Type_class A A' {dom : UR A A'} (P : A → Type) (Q : A' → Type) :=
 { transport_ :> Transportable P; ur_type :> ∀ x y (H:x ≈ y), P x ⋈ Q y }.

Definition URForall_Type A A' {HA : UR A A'} : UR (A → Type) (A' → Type) :=
  {| ur := fun P Q ⇒ URForall_Type_class A A' P Q |}.
```

This definition says that two predicates are in relation whenever they are in relation pointwise, and when `P` is transportable.

Using `Transportable`, we can instrument the definition of univalent relation on dependent products to improve effectiveness. More precisely, in the definition of the inverse function that defines the equivalence `(∀ x : A, B x) ≃ (∀ x : A', B' x)` we use the fact that `B` is transportable to change the dependency in `B` instead of pattern matching on the equality between the dependencies. This is possible because from $e_B : B \approx B'$, we know that `B` is transportable (thanks to the specialized definition `URForall_Type`).

6.4 Fully-Applied Predicates

Another scenario where the ad-hoc setting of univalent parametricity can be exploited to maximize effectiveness is when a type predicate is used fully-applied. For instance, consider the impredicative encoding of equality:

```
Definition iEq A (x y : A) := ∀ (P : A → Type), P x → P y.
```

and notice that the type predicate `P` above only appears in a fully-applied manner (`P x` and `P y`).

In order to show that `iEq` is univalently parametric, suppose we use the general proof that the dependent product is univalent (§ 3.3.3). Starting from two type families `P` and `Q` with a proof `H` that `∀ x x' : x ≈ x' → P x ≈ Q x'`, we first apply functional extensionality to deduce `P = Q`, and then rewrite `P` into `Q` in the type `P x → P y`. This use of functional extensionality makes the induced equivalence non effective.

However, because the type family `P` in `iEq` always appears fully applied, we need not rely on functional extensionality. The hypothesis `H` is enough to rewrite `P x → P y` into `Q x → Q y` directly.

6.5 Characterizing an Effective Fragment

The techniques described above go a long way in supporting useful applications of univalent transport. The open-endedness of the type class framework allows programmers (us included) to progressively augment effectiveness of univalent transport in Coq by building an extensive library of type class instances that exploit the specificities of certain transport scenarios.

There are two limits to this approach, however. First, there are equivalences that cannot be effective, because they imply either univalence or functional extensionality. Indeed, it is possible to lift the reflexivity proof that $\mathbb{N} = \mathbb{N}$ to a term of type $\mathbb{N} = \mathtt{N}$ using the equivalence between $\mathbb{N}$ and `N`. But no closed term inhabits $\mathbb{N} = \mathtt{N}$ in CIC.

Second, instead of letting programmers realize that a transported function is ineffective when using it, and then possibly trying to figure out whether a specific instance could address the problem, programmers might want a predictable (even though conservative) syntactic approach to *anticipate* whether a given transported function will be effective or not.

Fortunately, we can easily characterize a useful subset of CIC for which effectiveness of univalent transport is guaranteed to succeed. This fragment is characterized by two criteria:

(1) All indexed inductive families involved in the transport must be of types that have decidable equality.
(2) Types involved in the transport cannot quantify over functions whose codomain is the universe.

The first criterion is clear from the discussion in § 6.2. It is important here to recall the distinction between parameters and indices; the criterion only restricts indices, not parameters.

The second criterion aims at making sure that the only dependencies that are subject to transport come from the use of indexed inductive types. In particular, it rules out in the definition of the type arbitrary `A → Type` predicates, such as used in impredicative encodings like `iEq` above.[14]

This fragment is sufficiently expressive to be interesting: it contains System F_ω, extended with indexed inductive types with a decidable equality, which is often sufficient for certified programming applications of Coq.

Note that the types of inductive principles themselves are outside of the fragment. For instance, `ℕ_rect : ∀ (P : ℕ → Type), ...` quantifies over a type predicate. This means that if we transport `ℕ_rect` to an inductive principle on binary numbers `N`, it may not be effective. But any use of `ℕ_rect` to build a type is in the fragment. As an example, the following type, which expresses the commutativity of addition, lives in the fragment:

```
∀ (n m:ℕ), ℕ_rect (fun _ ⇒ ℕ) m (fun _ res ⇒ S res) n =
           ℕ_rect (fun _ ⇒ ℕ) n (fun _ res ⇒ S res) m
```

Example. Let us go back to the library example from § 1. The library itself is a record, which is effectively univalently parametric (§ 4.2). Like induction principles, `Lib` is outside of the fragment as it quantifies over `Type → ℕ → Type`, but any use of it (like our instantiation with vectors) is in the fragment.

The `head` and `map` attributes are standard functions that only quantify on types, not type predicates. Also, the vector inductive type ranges over indices that have decidable equality. Therefore, we definitely know that the transported library will be an actual record with effective implementations of `head` and `map`. Indeed:

```
Eval compute in liblist.(map) S [[1;2]].
 = [[2; 3]]
 : {l : list ℕ & length l = 2}
```

Also, the correctness property `prop` has automatically been ported to lists:

```
Check liblist.(prop _).
  : ∀ (n : ℕ) (A B : Type) (f : A → B) (v : {l : list A & length l = S n}),
    head liblist (map liblist f v) = f (head liblist v)
```

However, `liblist.prop` does not compute in general because it states an equality between elements of `B`, and we know nothing about `B`; in particular, it may not have a decidable equality. Arguably, since the property is morally computationally irrelevant for a typical certified programming task, there is not much interest in such effectiveness. But if desired, the only solution is to define `prop` such that `B` is a type with decidable equality; then the induced equivalence produces a proposition `liblist_prop` that is effective.

[14] As explained in § 6.4, the particular case of `iEq` can be addressed through ad-hoc polymorphism because the type predicate is used fully-applied.

6.6 A Note on Efficiency

Finally, we observe that effective proofs of univalent parametricity are not computationally equivalent in practice. For instance, as explained at the end of § 4.4, we could define univalence of `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 `largeVector`: `Vector.t` $\mathbb{N}$ 20 of size 20 has been defined. Using our framework, it is possible to define automatically

```
Definition largeVectorN : Vector.t N 20 := ↑ largeVector.
```

However, evaluating (the compiled version of) `largeVectorN` takes around 3 seconds on a quad-core Intel Core i5 3.5GHz machine. Conversely, the evaluation of the same term obtained by transporting along an equivalence that reasons directly on vector takes less than 1 *milli*second! Likewise, transporting functions that operate on vectors of $\mathbb{N}$ to functions that operate on vectors of `N` will be much more efficient if the direct equivalence is used.

7 RELATED WORK

Type theories. Homotopy Type Theory [25], 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, and supporting specialized proofs to avoid them in certain scenarios.

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 dependent 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 Extensional Type Theory, Krishnaswami and Dreyer [16] 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 § 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 $\textsc{Type}_i : \textsc{Type}_{i+1}$ 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 [18, 19] 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 `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. Moreover, the style they advocate prevents doing local transport and rather requires working with interfaces and applying parametricity in a second step, while we can avoid that thanks to our limitation to transport by equivalences.

Haftmann *et al.* [14] 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 [17] 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 [15] 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 [27] present an algorithm and plugin to transport theorems along isomorphisms in Coq similar to that of Huffman and Kunčar [15]. 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 $\forall x_1 \dots x_n,\ R(x_1 \dots x_n) \implies R'(f(x_1) \dots f(x_n))$, 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.

8 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 CC_ω 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 functional extensionality) axiom in a few places, univalent transport is not generally effective, but it is for a large fragment of the theory. We exploit the open-ended, ad-hoc formulation of univalent parametricity to provide effective proofs whenever possible. In addition, we identify a useful fragment of CIC for which univalent transport is guaranteed to be effective. In practice, this means that our Coq framework can readily be used to transport certified libraries and theories along type equivalences. Also, the open-endedness of the type class framework allows 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 proofs to provide.

To conclude, we observe that while parametric polymorphism is *a priori* invariance of constructions by potentially-unrelated types, univalent transport is *a posteriori* invariance of constructions by equivalent types. This work paves the way to easier-to-use environments for certified programming by supporting seamless programming and proving modulo equivalences.

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

[1] Thorsten Altenkirch and Ambrus Kaposi. Towards a cubical type theory without an interval. Accepted for publication in LIPIcs, 2017.

[2] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *Proceedings of the Workshop on Programming Languages meets Program Verification (PLPV 2007)*, pages 57–68, 2007.

[3] Abhishek Anand and Greg Morrisett. Revisiting parametricity: Inductives and uniformity of propositions. *CoRR*, abs/1705.01163, 2017.

[4] John Bacon. The untenability of genera. *Logique et Analyse*, 17(65/66):197–208, jan-apr 1974.

[5] Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau, and Bas Spitters. The hott library: A formalization of homotopy type theory in coq. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, pages 164–172, New York, NY, USA, 2017. ACM.

[6] Jean-Philippe Bernardy, Thierry Coquand, and Guilhem Moulin. A presheaf model of parametric type theory. *Electronic Notes in Theoretical Computer Science*, 319:67–82, 2015.

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

[8] S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors. *Proceedings of the 4th International Conference on Interactive Theorem Proving (ITP 2013)*, volume 7998 of *Lecture Notes in Computer Science*. Springer-Verlag, 2013.

[9] Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. The next 700 syntactical models of type theory. In *Certified Programs and Proofs (CPP 2017)*, pages 182 – 194, Paris, France, January 2017.

[10] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: a constructive interpretation of the univalence axiom. Accepted for publication in LIPIcs, October 2016.

[11] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free! In G. Gonthier and M. Norrish, editors, *Proceedings of the International Conference on Certified Programming and Proofs (CPP 2013)*, volume 8307 of *Lecture Notes in Computer Science*, pages 147–162. Springer-Verlag, 2013.
[12] The Coq Development Team. *The* Coq *proof assistant reference manual*. 2016. Version 8.6.
[13] Nicola Gambino and Martin Hyland. Wellfounded trees and dependent polynomial functors. In *Proceedings of Types for Proofs and Programs (TYPES 2003)*, volume 3085 of *Lecture Notes in Computer Science*, pages 210–225. Springer-Verlag, 2004.
[14] Florian Haftmann, Alexander Krauss, Ondřej Kunčar, and Tobias Nipkow. Data refinement in Isabelle/HOL. In Blazy et al. [8], pages 100–115.
[15] Brian Huffman and Ondřej Kunčar. Lifting and Transfer: A modular design for quotients in Isabelle/HOL. In *Proceedings of the 3rd International Conference on Certified Programs and Proofs (CPP 2013)*, pages 131–146, Melbourne, Australia, December 2013. Springer-Verlag.
[16] Neelakantan R. Krishnaswami and Derek Dreyer. Internalizing relational parametricity in the extensional calculus of constructions. In *Proceedings of the Conference for Computer Science Logic (CSL 2013)*, pages 432–451, 2013.
[17] Peter Lammich. Automatic data refinement. In Blazy et al. [8], pages 84–99.
[18] Nicolas Magaud. Changing data representation within the Coq system. In D. Basin and B. Wolff, editors, *International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2758 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.
[19] Nicolas Magaud and Yves Bertot. Changing data structures in type theory: A study of natural numbers. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *International Workshop on Types for Proofs and Programs (TYPES 2000)*, volume 2277 of *Lecture Notes in Computer Science*, pages 181–196. Springer-Verlag, 2000.
[20] Per Martin-Löf. An intuitionistic theory of types, 1971. Unpublished manuscript.
[21] Christine Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. In Bruno Woltzenlogel Paleo and David Delahaye, editors, *All about Proofs, Proofs for All*, volume 55 of *Studies in Logic (Mathematical logic and foundations)*. January 2015.
[22] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the 11th ACM SIGPLAN Conference on Functional Programming (ICFP 2006)*, pages 50–61, Portland, Oregon, USA, September 2006. ACM Press.
[23] John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.
[24] Matthieu Sozeau and Nicolas Oury. First-class type classes. In *Proceedings of the 21st International Conference on Theorem Proving in Higher-Order Logics*, pages 278–293, Montreal, Canada, August 2008.
[25] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013.
[26] Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, 1989.
[27] Theo Zimmermann and Hugo Herbelin. Automatic and transparent transfer of theorems along isomorphisms in the Coq proof assistant. arXiv:1505.05028v4, 2015.