Equivalences for Free!

Nicolas Tabareau, Éric Tanter, Matthieu Sozeau

Equivalences for Free!

Univalent Parametricity for Effective Transport

NICOLAS TABAREAU, Gallinette Project-Team, Inria
ÉRIC TANTER, PLEIAD Lab, Computer Science Department (DCC), University of Chile
MATTHIEU SOZEAU, Pi.R2 Project-Team, Inria and IRIF

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

ACM Reference format:
Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. 2017. Equivalences for Free!. 1, 1, Article 1 (July 2017), 27 pages.
https://doi.org/0000001.0000001

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 [2000] 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 [Huffman and Kunčar 2013] or Coq [Cohen et al. 2013; Zimmermann and Herbelin 2015]. At their core, most of these approaches build on parametricity [Reynolds 1983] and its potential for free theorems [Wadler 1989] in order to obtain results such as data refinements for free [Cohen et al. 2013] and proofs for free [Bernardy et al. 2012]. 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.

2017. XXXX-XXXX/2017/7-ART1 $15.00
https://doi.org/0000001.0000001

Univalence [Univalent Foundations Program 2013] 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.

Univalent transport in action. As a simple illustration, 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.

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

```
Definition liblist: Lib (fun A n ⇒ {l: list A & length l = n}) := ↑ libvec.
```

This way, the user gets a library on lists that is usable out of the box, and correct by construction. In particular, the proof of `prop` has been automatically converted to establish the property over lists.

This desirable scenario is however unsupported by existing approaches.

Univalence and computation. The scenario above assumes that univalent transport is *effective*: given a closed term of type $P\ A$, 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* [Univalent Foundations Program 2013]. The univalence axiom can be used in particular to establish what we hereby call the *Indiscernibility of Equivalents*,[1] hereafter IoE, 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 the work of Cohen et al. [2016] on cubical type theory (CubicalTT). 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. Because of the challenges involved, CubicalTT

[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 [1974].

does not currently support indexed inductive types (such as `Vector`), and it is currently unclear to what extent they can be supported in that setting.

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

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 [Reynolds 1983] that we coin *univalent parametricity*. By supporting the justification of univalent parametricity per type constructors, we can, on a case-by-case basis, try to avoid using axioms altogether, or at least push the use of axioms out of the computationally-relevant parts, hence supporting effective univalent transport for a large class of programs. More precisely:

- We introduce *univalent parametricity* as a strengthening of parametricity to ensure preservation of equivalences (Section 3).
- We provide a logical relation for univalent parametricity defined over type constructors (Section 3.1). The principle of indiscernibility of equivalents (IoE) for a type constructor amounts to the fundamental property of this logical relation (Section 3.2). We prove that each type constructor of the Calculus of Constructions with universes CC_ω is univalently parametric, identifying in each case the necessary assumptions (Section 3.3).
- We also define univalent parametricity through a translation in the style of Bernardy et al. [2012], which allows us to prove an abstraction theorem that entails that all terms of CC_ω are univalently parametric (Section 3.4).
- We develop univalent parametricity for inductive types (Section 4), addressing dependent sums and dependent records, and then dealing separately with parameterized type families and indexed type families.
- The logical relation for univalent parametricity serves as the foundation for an ad-hoc realization of univalent parametricity in Coq with type classes [Sozeau and Oury 2008], which is readily applicable to existing Coq developments, such as our introductory example (Section 5).
- We discuss the impact of the proofs of univalent parametricity on the effectiveness and efficiency of the induced univalent transport (Section 6).

The technical content of this work is fully formalized and proven in Coq (v8.6), including the translation and its properties, the type class framework and its instances, as well as the examples. The Coq source files are attached 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 developed in the following section.

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*, or IoE.

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 IoE above 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.

Because effectiveness is a central concern of this work, we briefly define what it means for a function to be effective, and by extension, for an equivalence.

Definition 2.5 (Effective function). A function $f : A \to B$ is said to be *effective* iff given a closed term $a : A$ as input, it produces a closed term $f\ a : B$ as output.

In other words, an effective function ensures *canonicity*, or, it does not get stuck by using an axiom. Conversely, a function that uses an axiom and hence "does not compute" is called *ineffective*.

From the above definition we can derive the notion of an effective equivalence.

Definition 2.6 (Effective equivalence). 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 (IoE) 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 [Bauer et al. 2017] . For instance, for the dependent product and sum, 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_sigma : ∀ A B (P : A → Type) (Q : B → Type)
                                 (e : A ≃ B) (e' : ∀ a, P a ≃ Q (↑ a)), sigT P ≃ sigT Q.
```

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 sums. By exploiting the functoriality of the dependent sum and products with respect to equivalence, `equiv_functor_sigma` 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_functor_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_functor_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 what could be called "the transport hell".

Escaping the transport hell. Looking back at `equiv_functor_∀` and `equiv_functor_sigma`, 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 was used to simplify the formulation of Observational Type Theory [Altenkirch et al. 2007].

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 IoE (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 [Reynolds 1983]. 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 [Reynolds 1983]. Recently, Bernardy et al. [2012] generalized the approach to pure type systems, including 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 the impredicative sort Prop, a hierarchy of universes $\textsc{Type}_i$, variables, applications, lambda expressions and dependent products:

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

Its typing rules are standard, and hence omitted here—see [Paulin-Mohring 2015] for a recent presentation.

Binary 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

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

Fig. 1. Parametricity translation for CC_ω, adapted from Bernardy et al. [2012]

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 Bernardy et al. [2012]. For the universe $\textsc{Type}_i$, the translation is naturally defined as (arbitrary) binary relations on types. Similarly, the translation of Prop is the type of binary relations on propositions. 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 Bernardy et al. [2012], 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 2.7 (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) [Bernardy et al. 2012]. However, they are insufficient to ensure preservation of equivalences (IoE). 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 the discussion of inductive types to Section 4.

We first define a *univalent logical relation* as a type-indexed logical relation on all the type constructors of CC_ω (Section 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 (Section 3.2). We discuss in Section 3.3 the proofs that each type constructor is univalently parametric, paying attention to the potential use of axioms.

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, Section 3.4 presents a translation for univalent parametricity in the style of Bernardy et al. [2012]. For type constructors, the translation appeals to proof terms previously introduced in Section 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 (Section 5).

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 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

$$
\begin{aligned}
A \approx B : \textsc{Type}_i \bowtie \textsc{Type}_i &\triangleq A \bowtie B \ \wedge \ A \simeq B \ \wedge \\
&\quad \forall a : A, b : B, (a \approx b : A \bowtie B) \simeq (a = \uparrow b) \\
A \approx B : \textsc{Prop} \bowtie \textsc{Prop} &\triangleq A \bowtie B \ \wedge \ A \simeq B \ \wedge \\
&\quad \forall a : A, b : B, (a \approx b : A \bowtie B) \simeq (a = \uparrow b) \\
P \approx Q : A \to \textsc{Type}_i \bowtie B \to \textsc{Type}_i &\triangleq A \bowtie B \ \wedge \\
&\quad \forall a : A, \forall b : B, a \approx b : A \bowtie B \implies P\ a \approx Q\ b : \textsc{Type}_i \bowtie \textsc{Type}_i \\
f \approx g : \Pi a : A.P\ a \bowtie \Pi b : B.Q\ b &\triangleq P \approx Q : A \to \textsc{Type}_i \bowtie B \to \textsc{Type}_i \ \wedge \\
&\quad \forall a : A, \forall b : B, a \approx b : A \bowtie B \implies f\ a \approx g\ b : P\ a \bowtie Q\ b
\end{aligned}
$$

Fig. 2. Univalent relation for CC_ω

insufficient because there is no connection between the two notions. This in particular implies that the identity type, which defines the notion of equality, will not satisfy the fundamental property of the logical relation. For this, we need to additionally demand that the heterogeneous relation *coincides with propositional equality* once the values are at the same type.

Formally, we introduce a logical relation for univalent parametricity, called the *univalent relation*, defined in 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. 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 (Section 2.3). The fundamental difference is that in our approach, this arbitrary choice is deferred as late as possible, *i.e.* when we *do* need to know more about the univalent relation.

As alluded to above, the coherence condition is used in particular in the proof that the identity type is related to itself. In that case, we need to prove that

$$\forall A\ B : \textsc{Type},\ \forall a\ a' : A,\ \forall\ b\ b' : B',\ a \approx b : A \bowtie B \ \wedge \ a' \approx b' : A \bowtie B \implies a = a' \simeq b = b'$$

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

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

The univalent relation for the sort $\textsc{Prop}$ is the same as for $\textsc{Type}_i$: the propositions must be equivalent, and related by a relation that is coherent with equality.

On the other type constructors, the univalent logical relation is similar to parametricity. In particular, at type families $A \to \textsc{Type}_i$ and $B \to \textsc{Type}_i$, 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 $\textsc{Type}_i$. In the same way, at dependent products $\Pi a : A.P\ a$ 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 $\mathsf{Univ}(x)$, 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 Section 3.4 using a translation. But we can already state and prove an important property: that a univalently parametric type constructor preserves type equivalences.

Proposition 3.2 (Univalent constructor satisfies IoE). *Let $P : \mathit{Type}_i \to \mathit{Type}_i$ be a univalently parametric constructor,* i.e. *$\mathit{Univ}(P)$, then $A \simeq B \implies P\ A \simeq P\ B$.*

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

$$\forall A\ B : \mathrm{Type}_i,\ A \approx B : \mathrm{Type}_i \bowtie \mathrm{Type}_i \implies P\ A \approx P\ B : \mathrm{Type}_i \bowtie \mathrm{Type}_i$$

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 $A \approx B : \mathrm{Type}_i \bowtie \mathrm{Type}_i$. Therefore, $P\ A \approx P\ B : \mathrm{Type}_i \bowtie \mathrm{Type}_i$, 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 $\mathsf{canon}(e) : A \approx B : \mathrm{Type}_i \bowtie \mathrm{Type}_i$ 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 eagerly imposed the use of this relation, we would back in transport hell as described in Section 2.3.

3.3 Type constructors are univalently parametric

We now prove that the universe Type_i, Prop, and the dependent product Π are univalent.

3.3.1 Type. $\mathsf{Univ}(\mathrm{Type}_i)$ 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. *$\mathit{Univ}(\mathit{Type}_i)$ is inhabited.*

Proof. First, we need to define a relation between Type_i and Type_i. By a fixpoint argument, it has to be $\mathrm{Type}_i \bowtie \mathrm{Type}_i$. We also need to provide an equivalence $\mathrm{Type}_i \simeq \mathrm{Type}_i$; we simply take the identity equivalence $\mathrm{id}_{\mathrm{Type}_i}$. Finally, we need to prove that the relation is coherent with equality, that is, we need to exhibit a term $\mathsf{univ}_{\mathrm{Type}_i}$ such that:

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

For the function from $A = B$ to $A \approx B : \mathrm{Type}_i \bowtie \mathrm{Type}_i$, by induction on equality, it is sufficient to provide the canonical inhabitant $\mathsf{canon}(\mathrm{id}_A) : A \approx A : \mathrm{Type}_i \bowtie \mathrm{Type}_i$ associated to the identity equivalence, as used in the proof of Proposition 3.2.

The converse direction follows directly from univalence because $A \approx B : \mathrm{Type}_i \bowtie \mathrm{Type}_i$ by definition implies $A \simeq B$, which by the univalence axiom implies $A = B$.

Finally, the proofs that these two functions are indeed inverse of each other (section, retraction) make use of the section and the retraction of the equivalence postulated by the univalence axiom, together with lemmas about decomposition of equality over $\text{Type}_i \bowtie \text{Type}_i$ and commutation of transports; the interested reader can consult the Coq development.

□

3.3.2 Prop. In our definition, Prop is treated in the same way as Type_i because $\text{Prop} : \text{Type}_i$ 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 $\text{Univ}(\text{Type}_i)$ 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 be deduced from univalence and proof irrelevance for Prop but is stronger than just univalence. As we are looking for the minimal amount of axioms needed for establishing univalent parametricity, we do not make use of this stronger axiom.

Note that exploiting the fact Prop is proof irrelevant, $\text{Prop} \bowtie \text{Prop}$ boils down to

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

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 Morrisett [2017]—more details in Section 7.

3.3.3 Dependent product. We now show that the dependent product 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.[2]

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

Proof. Univ(Π) unfolds to

$$\Pi A\ B\ P\ Q.\ A \approx B : \text{Type}_i \bowtie \text{Type}_i \to P \approx Q : A \to \text{Type}_i \bowtie B \to \text{Type}_i \to$$
$$\Pi(a : A).\ P\ a \approx \Pi(b : B).\ Q\ b : \text{Type}_i \bowtie \text{Type}_i$$

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 [Univalent Foundations Program 2013], namely `equiv_functor_∀` in the Coq HoTT library [Bauer et al. 2017], which was already introduced

[2]This result is folklore; see for instance:
https://homotopytypetheory.org/2014/02/17/another-proof-that-univalence-implies-function-extensionality/.

in Section 2.3. This lemma is effective for the two transport functions but requires functional extensionality in the proof that they form an equivalence.[3]

We note the resulting term Equiv_Π, with:

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

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

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

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 described in Section 2.3. Indeed, by composing with the equivalence induced by functional extensionality (and after some healthy HoTT yoga), this amounts to prove

$$\mathsf{univ}_\Pi : \Pi ab(e : a \approx b : A \bowtie B), (f\ a \approx \uparrow(g\ b) : P\ a \bowtie Q\ b) \simeq (f\ a \approx (\uparrow g)\ b : P\ a \bowtie Q\ b)$$

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 Section 2.4). Figure 3 shows how to extend the relational parametricity translation to force the heterogeneous relation defined between two types to correspond to a type equivalence with the coherence condition. Note that the translation does not target CC_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 Section 3.1, for Type_i, following Figure 2 but switching to the type theoretical notation, we want to set[4]

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

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* (*i.e.* right of the ":"), translated as $[\![T]\!]_u$.[6] The fundamental property on Type_i enforces the definition

[3] The definition of the inverse function requires using the retraction, and the proof that it forms a proper equivalence requires the adjunction condition (Section 2.1). This means that the dependent product 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 sum, 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.

[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. [2017]. For instance, to prove the independence of univalence with CIC, they use a translation that associates a Boolean to any type, *e.g.* $[\text{Type}_i] = (\text{Type}_i \times \mathbb{B}, \text{true})$. 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.

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

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

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

Fig. 3. Univalent parametricity translation for CC_ω

of the relation, equivalence and coherence on a type T to be deduced from $[T]_u$ respectively as

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

The 3-tuples for $\textsc{Type}_i$ and dependent products are precisely given by the fact that they are in the diagonal of the univalent relation, as proved in Section 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 following corollary, which 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, $[a]_u : a \approx a : A \bowtie A$. □

Finally, note that although the translation for products is defined for two terms a and a' of respective types A and A', A' is not any arbitrary type: it is the result of duplication with renaming applied to A (Section 2.4); likewise, a' is a renamed duplicate of a. Additionally, a and a' are expected to be related according to the interpretation of the *single* type A. This is why the univalent logical

relation of Figure 2 is more general than the univalent parametricity translation: it can describe relations between terms of arbitrarily different types, as long as some equivalence can be exhibited. For instance, we can relate naturals $\mathbb{N}$ and binary naturals N, *i.e.* $\mathbb{N} \approx \mathtt{N} : \textsc{Type} \bowtie \textsc{Type}$.

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 an inductive type I (with constructors I_cons_i and elimination I_rect) is univalent and thus being able to extend the abstraction theorem of CC_ω, one needs to prove that $\mathrm{Univ}(I)$, $\mathrm{Univ}(I\_cons_i)$ and $\mathrm{Univ}(I\_rect)$. Of course, using the univalence axiom from the outset, it is always possible to show that an inductive is univalent; but we are interested in definitions that are as effective as possible, and therefore rely on axioms only when strictly necessary.

In this section, we first consider the case of non-recursive inductive types without indices: dependent sums and records (Section 4.1). We show that the fact that dependent sums—and thus records—are univalent does not need any axiom. We then consider the case of parameterized inductive families in Section 4.2. We show that the type of lists is univalent and sketch a general construction for such families. Finally, Section 4.3 deals with indexed inductive families, such as Generalized Algebraic Data Types [Peyton Jones et al. 2006] and the prototypical case of identity types. We show that, in general, proving the univalent parametricity of indexed families requires using the coherence condition. Because the coherence conditions are the main point of ineffectiveness (*i.e.* for the universe and dependent functions), the effectiveness of univalent transport for indexed inductive families depends on the type of the indices. We come back to effectiveness in Section 6.

4.1 Dependent sums and records

We show that dependent sums are univalent, and explain how this result generalizes to dependent records, exploiting the fact that a dependent record is equivalent to a telescope of dependent sums.

4.1.1 Dependent sums. In CIC, dependent sums 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 sum 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 sum types are part of the syntax [Martin-Löf 1971].

[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 sums

Proposition 4.1. *Univ*(Σ) *is inhabited.*

Proof. Univ(Σ) unfolds to

$$\Pi A\ B\ P\ Q.\ A \approx B : \text{Type}_i \bowtie \text{Type}_i \to P \approx Q : A \to \text{Type}_i \bowtie B \to \text{Type}_i \to \Sigma a : A.\ P\ a \approx \Sigma b : B.\ Q\ b : \text{Type}_i \bowtie \text{Type}_i$$

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.

Similarly to the case of dependent products, the proof that $\Sigma a : A.\ P\ a \simeq \Sigma b : B.\ Q\ b$ follows from a standard result of HoTT [Univalent Foundations Program 2013], namely `equiv_functor_sigma` in the Coq HoTT library [Bauer et al. 2017], which was already introduced in Section 2.3. Contrarily to the dependent product, which requires the functional extensionality axiom, this lemma does not require any axiom.

Again, 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:

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

This equivalence can be proven conveniently by composition of equivalences instead of building the equivalence explicitly with the transport functions and their associated section and retraction proofs. Specifically, we rely on a decomposition of equality for dependent sums:[9]

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

□

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 products. 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 sum.

4.1.2 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

[9]From now on, we occasionally write $x \approx y$ as a shortcut for $x \approx y : X \bowtie Y$, whenever the types of x and y, *i.e.* X and Y, are not necessary for the discussion.

```
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 $X : \mathrm{Type}_i$, $A\ B : X \to \mathrm{Type}_j$, 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 Coq as a tactic (Section 5).

4.2 Parameterized recursive inductive families

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

For instance, this can be used for the inductive definition of natural numbers:

```
Inductive ℕ : Type := O : ℕ | S : ℕ → ℕ
```

However, whenever an inductive type has parameters, the situation is more complex.

Parameters vs. indices. Because the distinction between *parameters* and *indices* for inductive types is important in our development of univalent parametricity, we first recall the basics.

A parameter is merely indicative that the type behaves *uniformly* with respect to the supplied argument. This is the case for instance for the inductive family `list` A defined at the beginning of this section: an inhabitant of that type has the same shape regardless of what A is: it is either the empty list, `nil`, or an element `a` of type A concatenated with a list `l` of type `list` A, *i.e.* `cons a l`. 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, indices may directly affect which values can inhabit the type. Consider for instance the inductive family `Vect A n`:

```
Inductive Vect A : ℕ → Type :=
 nil : Vect A 0
| cons : ∀ (h:A) (n:ℕ), Vect A n → Vect A (S n).
```

The index `0` indicates an empty vector while the index `1` indicates a vector with exactly one element. This means that from the knowledge of the value of the index, one can infer which constructor(s) may or may not have been used to create the value.

This section is devoted to parameters; the following section deals with indices.

Example. Let us develop the case of lists. First, we need to show that

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

The two transport functions of the equivalence `list` $A \simeq$ `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. [2012] on the *inductive-style translation*, 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 sums, 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. We believe that it is possible to generalize the above result 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 implementation, 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 (Section 5).

4.3 Indexed inductive families

CIC allows defining inductive types that are not parameterized, but rather indexed, like size-indexed vectors discussed previously. Another mainstream example are Generalized Algebraic Data Types [Peyton Jones et al. 2006], or GADTs for short, 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 what makes their univalence non-effective!

To see why, 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, by inspection of the constructors of `Expr`, there is no way to construct a value of type `Expr N`; therefore, the only way to obtain such a term is 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.

4.3.1 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

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

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 [Univalent Foundations Program 2013], namely `equiv_functor_eq` in the Coq HoTT library [Bauer et al. 2017], which was already introduced in Section 2.3.

The univalent relation for identity types is defined as

$$e_1 \approx e_2 : a =_A a' \bowtie b =_B b' \triangleq A \approx B : \text{Type}_i \bowtie \text{Type}_i \ \wedge\ a \approx b : A \bowtie B \ \wedge\ a' \approx b' : A \bowtie B \ \wedge\ e_1 =_{a=a'} \uparrow e_2$$

It makes use of the previous equivalence to transport the equality e_2 to an equality $\uparrow e_2$ of type $a =_A a'$ and then compare e_1 and $\uparrow e_2$ using the identity type on $a =_A a'$.

Note that we could also have used the inductive type that is obtained by applying parametricity to the identity type:

```
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.
```

We rather use a direct definition with equality because it involves less HoTT yoga, but we conjecture that using `UR_eq` should be possible as well.

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

$$\Pi(e\ e' : a = a').\ (e = e') \simeq (e = \uparrow\uparrow e')$$

which is a direct consequence of the fact that $\uparrow\uparrow e' = e'$. □

[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.

The proofs that `eq_refl` and `eq_rect` are univalent require higher coherence reasoning on identity types, as typically done in the HoTT library.

From indices to parameters. Finally, we observe that we can exploit the correspondence between an indexed inductive family and a subset of parameterized inductive family, established by Gambino and Hyland [2004], 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:

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

where `length` is the function that computes the length of a list. The `length` function is defined as

```
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 $\texttt{Vect}\ A\ n$. Additionally, the univalence of $\texttt{Vect}\ A\ n$ relies on the *coherence condition on natural numbers*, which is an effective equivalence. However, the obtained transport might not be very efficient as it implies going through lists—we come back to the efficiency of transport in Section 6.

Returning to the `Expr` example, we can construct the associated parameterized inductive type

```
Inductive Expr_p : Type :=
  I_p : ℕ → Expr_p
| B_p : 𝔹 → Expr_p
| Add_p : Expr_p → Expr_p → Expr_p
| Eq_p  : Expr_p → Expr_p → Expr_p.
```

and a fixpoint `index_Expr (E : Expr_p) (EType : Type) : Type`, computing for instance

```
index_Expr (Add_p E E') EType := (index_Expr E ℕ) * (index_Expr E' ℕ) * (EType = ℕ)
```

for the case of the addition $\texttt{Add}_p$. It is then possible to prove

```
Definition Expr_Expr_p (A:Set) : Expr A ≃ Σ E : Expr_p. index_Expr E A.
```

and conclude the univalence for `Expr` from the univalence of $\texttt{Expr}_p$ and of its elimination principle.

Of course, the situation has not been improved with respect to effectiveness because the fixpoint `index_Expr` uses equality on the universe `(EType = ℕ)`, which is ineffective.

5 UNIVALENT PARAMETRICITY IN COQ

The whole development of univalent parametricity exposed in this article has been formalized in the Coq system [Coq Development Team 2016], reusing several constructions from the HoTT library [Bauer et al. 2017]. We do not discuss the Coq formalization of the univalent parametricity translation of Section 3.4 here; instead, we focus on the shallow embedding of the univalent relation based on type class instances to define and automatically derive the univalence proofs of Coq constructions. We first introduce the core classes of the framework in Section 5.1, and then describe the instances for some type constructors in Section 5.2; Section 5.3 develops some examples, including the motivating example from the introduction.

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:[11]

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

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

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 }.
Infix "⋈" := UR_Type.
```

5.2 Univalent type constructors

The core of the development is devoted to the proofs that standard type constructors are univalently parametric notably Type, Prop, 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 Section 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. 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.

For dependent products, we set:

```
Definition UR_Forall A A' (B : A → Type) (B' : A' → Type) (Rdom: UR A A')
 (Rcod: ∀ 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 products 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.

More interestingly, the Equiv instance derived from this definition for dependent products 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 mentioned in Section 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 product 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 the other constructors, *i.e.* sigma types, the equality type, natural numbers and booleans with the canonical univalent relation, where we additionally prove the fundamental property for the eliminators. That is, we have many fundamental property lemmas such as

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

5.2.1 Record types. For record types, we reuse an idea used in the HoTT library [Bauer et al. 2017] that allows automated inferrence of type equivalence for records with their nested sigma type formulation. For example supposing we have a monoid interface defined as a record:

```
Record Monoid A := Build_Monoid {
 mon_e : A;
 mon_m : A → A → A;
 mon_unitL : ∀ x, mon_m x mon_e = x;
 mon_unitR : ∀ x, mon_m mon_e x = x;
 mon_α : ∀ x y z, mon_m x (mon_m y z) = mon_m (mon_m x y) z }.
```

We can express an equivalent type using nested sigma types, and automatically show the equivalence using the issig tactic provided by the HoTT library:

```
Instance issig_monoid (A : Type) :
 (Σ (e : A). Σ (m : A → A → A). Σ (uL : ∀ x, m x e = x). Σ (uR : ∀ x, m e x = x). ∀ x y z, m x (m y z) = m (m x y) z)
     ≃ Monoid A.
Proof. issig (Build_Monoid A) (mon_e A) (mon_m A) (mon_unitL A) (mon_unitR A) (mon_α A). Defined.
```

We can then reuse the properties of sigma types to show that Monoid is univalently parametric:

```
Definition FP_Monoid : Monoid ≈ Monoid.
Proof. univ_param_record. Defined.
```

Because the approach is generally applicable, our tactic univ_param_record can be used to automatically prove that any record type is univalently parametric.

5.2.2 Inductive types. We have developed some automation to derive the fundamental property for inductive families. For instance, we have developed a tactic `equiv_adt` that automatically infers the preservation of equivalence for algebraic data types. However, due to the restrictions described in Section 4.3, its applicability is limited. It is however always possible to use either the generic univalent relation using univalence, at the expense of effectiveness, or to construct the desired instance explicitly. We discuss these alternatives in Section 6.

5.3 Examples

With the framework in place, it is possible to realize the library example from Section 1. We first need to define that indexed vectors and lists refined with their length are related by the univalent relation (note the importance of the heterogeneity of the relation for user-defined applications).

Then, using automation for record types, we show that the `Lib` interface is univalently parametric:

```
Definition FP_Lib : Lib ≈ Lib.
Proof. univ_param_record. Defined.
```

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

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

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

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

```
Definition liblist : Lib (fun A n ⇒ {l: list A & length l = n}) := ↑ libvec.
```

We can check the effectiveness of this construction on a simple example:[12]

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

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

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

Another possible application of equivalences for free is the use of transport to switch between easy-to-reason-about and efficient representations, an approach known as data refinement [Cohen et al. 2013]. For instance, it is possible to show that (ordinary) natural numbers ℕ and binary natural numbers `N` are univalently related.

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.
```

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

[12] The notation `[[x;...;z]]` creates a list with a proof (by reflexivity) that it is of the appropriate size.

6 EFFECTIVENESS AND EFFICIENCY OF TRANSPORT

The fact that univalent parametricity is defined in an *ad hoc* manner, *i.e.* per constructor, allows us to follow different approaches. As a matter of fact, not all proofs of univalent parametricity are "equivalent" if we consider the effectiveness and efficiency of the transport that they entail. In particular, we have striven to maximize the effectiveness of transport by avoiding the use of axioms as much as possible, and when necessary, using them "as far back" as possible. Additionally, certain ways to establish univalent parametricity, even though effective, may yield transports that are more computationally expensive than others. We now briefly discuss and illustrate these concerns.

6.1 Effectiveness

Because we are in an (axiomatic) univalent theory, we can always immediately appeal to the univalence axiom to establish that a given constructor is univalently parametric. For instance, we can establish that the `list` constructor is univalently parametric by resorting to univalence:

```
Definition FL_List_Noneff : list ≈ list.
 intros A B e. destruct (e_inv (eq_to_equiv _ _) (equiv e)).
 apply Canonical_UR. apply Equiv_id.
Defined.
```

This proof starts by applying univalence to the assumption that `equiv e : A ≃ B` to rewrite `B` into `A`. The use of univalence is manifest in `e_inv (eq_to_equiv _ _)`, which produces equality based on an equivalence. At that stage, we know that `A ≈ A` holds, and need to establish `list A ≈ list A`. This follows directly by using the canonical univalent relation, because `list A` is equivalent to itself.

However, using this proof of the fundamental property of the univalent relation for `list` yields a transport that is not effective. To get effectiveness, we need a direct proof as explained in Section 4.2.

In Section 3.3, we have described proofs of univalent parametricity for $\textsc{Type}_i$, Prop, and the dependent product; in Section 4, we have addressed univalent parametricity for inductive types. The only direct use of axioms in these proofs are in the coherence property for the universe, which requires univalence, and in the section and retraction for dependent product, both of which require functional extensionality.

The use of an axiom in the coherence condition for the universe and the dependent products only raises effectiveness issues for indexed inductive families that have indices over such types, such as GADTs. Thus, a typical example of an ineffective equivalence inferred by our system is

$$(\forall\ f\colon \mathbb{N} \to \mathbb{N},\ f = f) \simeq (\forall\ f\colon \mathsf{N} \to \mathsf{N},\ f = f).$$

However, when indices are on a type for which the coherence condition is effective, we obtain an effective equivalence, *e.g.*

$$(\Sigma\ f : \mathbb{N} \to \mathbb{N}.\ \mathtt{vector}\ \mathbb{N}\ (f\ 0)) \simeq (\Sigma\ f : \mathbb{N} \to \mathbb{N}.\ \Sigma\ (l : \mathtt{list}\ \mathbb{N}\ .\ \mathtt{length}\ l = f\ 0)).$$

More importantly, the axiom in the proof of section/retraction of IoE (*i.e.* related input yield equivalent outputs, Proposition 3.2) for dependent product may lead to more subtle effectiveness issues. For instance, the proof of IoE for dependent sums requires the section/retraction of IoE on the domain for the inverse function, which is hence not effective. Fortunately, when a dependent sum is used in a strictly covariant position, the inverse function is not used. This is why the example above is effective despite its use of dependent sums indexed by functions.

Also, an equivalence may not be entirely effective, yet still compute on the relevant part. For instance, the equivalence

$$(\forall\ f : \{f : \mathsf{N} \to \mathsf{N}\ \&\ f\ (0\%\mathsf{N}) = (1\%\mathsf{N})\}, \mathsf{N}.) \simeq (\forall\ f : \{f : \mathbb{N} \to \mathbb{N}\ \&\ f\ 0 = 1\}, \mathbb{N}).$$

allows to transport the function `ff := fun f ⇒ f.1 0 + 1`, such that `↑ ff (N.succ; eq_refl)` computes actually 2. This is because the ineffective part of the equivalence is only on the property expressed by equality, which is not used to compute the resulting value.

Of course, it is possible to exhibit an ineffective equivalence, for instance by going higher-order; *e.g.* the equivalence between functions that take a function `f` and a vector of size `f 0`

```
∀ X : {f : (N → ℕ) & vector N (f 0%N)}, vector N (S (X.1 0%N)) ≃
∀ X : {f : (ℕ → ℕ) & vector ℕ (f 0)}, vector ℕ (S (X.1 0)).
```

is not effective.

6.2 Efficiency

Even effective proofs of univalent parametricity are not equivalent in practice.

Recall the proof of univalence of `Vect` *A n* through the equivalence with lists refined with a predicate on their length (Section 4.3). 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 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.

If instead we directly define the equivalence using our automated tactic `equiv_adt`

```
Instance Equiv_Vector A B (e:A ≃ B) n n' (en :n = n') : Vector.t A n ≃ Vector.t B n'.
Proof. equiv_adt (@Vector.t_rect _) (@Vector.nil _) (@Vector.cons _). Defined.
```

The evaluation of the same term obtained by transporting along `Equiv_Vector` takes less than 1 *milli*second! Likewise, transporting functions that operate on vectors of ℕ to functions that operate on vectors of `N` will be much more efficient if the direct equivalence is used.

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

7 RELATED WORK

Type theories. Homotopy Type Theory [Univalent Foundations Program 2013], and its embodiment in the HoTT library [Bauer et al. 2017] treat equality of types as equivalence. For regular datatypes (also know as homotopy sets or hSets), equivalence boils down to isomorphism, hence the existence of transports between the types. However, as univalence is considered as an axiom, any meaningful use of the equality type to transport terms along equivalences results in the use of a non-computational construction. In contrast we carefully delimit the effective equivalence preserving type constructors in our setting, pushing axioms as far as possible.

Cubical Type Theory [Cohen et al. 2016] 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. However, it is unclear how to extend this system with general indexed type families yet, as hinted in Section 4.3.

Observational Type Theory (OTT, [Altenkirch et al. 2007]) 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 [Bernardy et al. 2015] 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 actually effectively univalent.

For Extensional Type Theory, Krishnaswami and Dreyer [2013] 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 Morrisett [2017] extends the logical relation at propositions to force that related propositions are logically equivalent. It can be seen as a degenerate case of our extension which forces related types to be equivalent, as equivalence boils down to logical equivalence on propositions (see Section 3.3 for a more detailed explanation). However the translations differ in other aspects, while our translation requires the univalence axiom theirs assumes proof irrelevance and the K axiom, and does not treat the type hierarchy. Our solution to the fixpoint arising from interpreting $\textsc{Type}_i : \textsc{Type}_{i+1}$ 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 axiom 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 [2003]; Magaud and Bertot [2000] 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 was limited to isomorphism and implemented externally as a plugin. The technique was rather invasive in the sense that it supported the transport of proof terms that used 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 breaks down in presence of type dependency.

In CoqEAL [Cohen et al. 2013] refinement is allowed from proof-oriented data types to efficiency-oriented ones, relying on generic programming for the computational part and automating the transport of theorems and proofs. They not only deal with isomorphisms, but also quotients, and even partial quotients, which we cannot handle. Still, they can and do exploit parametricity for generating proofs but they do not support general dependent types, only parametric polymorphism.

Haftmann et al. [2013] 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 [Lammich 2013] 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 [2013] 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 [2015] present an algorithm and plugin to transport theorems along isomorphisms in Coq similar to Huffman and Kunčar [2013]. 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 function extensionality) axiom in few places, effectiveness is still an issue, but we have been very careful to push axioms as far back as possible to get maximal effectiveness of the basic equivalences we provide. In practice, this means that our Coq framework can readily be used to transport certified libraries and theories along type equivalences.

We identify that indexed inductive families, in particular GADTs, raise a challenging effectiveness issue, depending on the types of the indices. This issue applies to all univalent type theories, including Cubical Type Theory. While giving a computational interpretation of univalence for inductive types in full generality, if at all possible, is still an open research question, our work contributes by providing an extensible framework on top of which to experiment. Indeed, the open-ended nature of the type class framework allows users to extend and improve on our current definitions, for instance by providing more effective or more efficient instances for specific type constructors.

ACKNOWLEDGMENTS

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

REFERENCES

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

Abhishek Anand and Greg Morrisett. 2017. Revisiting Parametricity: Inductives and Uniformity of Propositions. *CoRR* abs/1705.01163 (2017). http://arxiv.org/abs/1705.01163

John Bacon. 1974. The Untenability of Genera. *Logique et Analyse* 17, 65/66 (jan-apr 1974), 197–208.

Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau, and Bas Spitters. 2017. 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)*. ACM, New York, NY, USA, 164–172. DOI: http://dx.doi.org/10.1145/3018610.3018615

Jean-Philippe Bernardy, Thierry Coquand, and Guilhem Moulin. 2015. A Presheaf Model of Parametric Type Theory. *Electronic Notes in Theoretical Computer Science* 319 (2015), 67–82.

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

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

Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. 2017. The next 700 syntactical models of type theory. In *Certified Programs and Proofs (CPP 2017)*. Paris, France, 182 – 194. DOI: http://dx.doi.org/10.1145/3018610.3018620

Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2016. Cubical Type Theory: a Constructive Interpretation of the Univalence Axiom. arXiv:1611.02108. (2016).

Cyril Cohen, Maxime Dénès, and Anders Mörtberg. 2013. Refinements for Free!. In *Proceedings of the International Conference on Certified Programming and Proofs (CPP 2013) (Lecture Notes in Computer Science)*, G. Gonthier and M. Norrish (Eds.), Vol. 8307. Springer-Verlag, 147–162.

The Coq Development Team. 2016. *The* Coq *proof assistant reference manual*. http://coq.inria.fr Version 8.6.

Nicola Gambino and Martin Hyland. 2004. Wellfounded trees and dependent polynomial functors. In *Proceedings of Types for Proofs and Programs (TYPES 2003) (Lecture Notes in Computer Science)*, Vol. 3085. Springer-Verlag, 210–225.

Florian Haftmann, Alexander Krauss, Ondřej Kunčar, and Tobias Nipkow. 2013. Data Refinement in Isabelle/HOL, See [Blazy et al. 2013], 100–115.

Brian Huffman and Ondřej Kunčar. 2013. 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)*. Springer-Verlag, Melbourne, Australia, 131–146.

Neelakantan R. Krishnaswami and Derek Dreyer. 2013. Internalizing Relational Parametricity in the Extensional Calculus of Constructions. In *Proceedings of the Conference for Computer Science Logic (CSL 2013)*. 432–451.

Peter Lammich. 2013. Automatic Data Refinement, See [Blazy et al. 2013], 84–99.

Nicolas Magaud. 2003. Changing Data Representation within the Coq system. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003) (Lecture Notes in Computer Science)*, D. Basin and B. Wolff (Eds.), Vol. 2758. Springer-Verlag.

Nicolas Magaud and Yves Bertot. 2000. Changing Data Structures in Type Theory: A Study of Natural Numbers. In *International Workshop on Types for Proofs and Programs (TYPES 2000) (Lecture Notes in Computer Science)*, P. Callaghan, Z. Luo, J. McKinna, and R. Pollack (Eds.), Vol. 2277. Springer-Verlag, 181–196.

Per Martin-Löf. 1971. An Intuitionistic Theory of Types. (1971). Unpublished manuscript.

Christine Paulin-Mohring. 2015. Introduction to the Calculus of Inductive Constructions. In *All about Proofs, Proofs for All*, Bruno Woltzenlogel Paleo and David Delahaye (Eds.). Studies in Logic (Mathematical logic and foundations), Vol. 55.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple unification-based type inference for GADTs. In *Proceedings of the 11th ACM SIGPLAN Conference on Functional Programming (ICFP 2006)*. ACM Press, Portland, Oregon, USA, 50–61.

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *IFIP Congress*. 513–523.

Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *Proceedings of the 21st International Conference on Theorem Proving in Higher-Order Logics*. Montreal, Canada, 278–293.

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

Philip Wadler. 1989. Theorems for Free!. In *Functional Programming Languages and Computer Architecture*. ACM Press, 347–359.

Theo Zimmermann and Hugo Herbelin. 2015. Automatic and Transparent Transfer of Theorems along Isomorphisms in the Coq Proof Assistant. arXiv:1505.05028v4. (2015).