






Trocq: Proof Transfer for Free, With or Without Univalence^{*}

Cyril Cohen¹()^(✉), Enzo Crance^{2,3}()^(✉), and Assia Mahboubi^{2,4}()^(✉)

¹ Université Côte d’Azur, Inria, Sophia Antipolis, France

`Cyril.Cohen@inria.fr`

² Nantes Université, École Centrale Nantes, CNRS, INRIA, LS2N, UMR 6004,
Nantes, France

`{Enzo.Crance, Assia.Mahboubi}@inria.fr`

³ Mitsubishi Electric R&D Centre Europe, Rennes, France

⁴ Vrije Universiteit Amsterdam, Amsterdam, The Netherlands

Abstract. This article presents TROCQ, a new proof transfer framework for dependent type theory. TROCQ is based on a novel formulation of type equivalence, used to generalize the univalent parametricity translation. This framework takes care of avoiding dependency on the axiom of univalence when possible, and may be used with more relations than just equivalences. We have implemented a corresponding plugin for the Coq interactive theorem prover, in the Coq-Elpi meta-language.

Keywords: Parametricity, Representation independence, Univalence, Proof assistants, Proof transfer

1 Introduction

Formalizing mathematics provides every object and statement of the mathematical literature with an explicit data structure, in a certain choice of foundational formalism. As one would expect, several such explicit representations are most often needed for a same mathematical concept. Sometimes, these different choices are already made explicit on paper: multivariate polynomials can for instance be represented as lists of coefficient-monomial pairs, *e.g.*, when computing Gröbner bases, but also as univariate polynomials with polynomial coefficients, *e.g.*, for the purpose of projecting algebraic varieties. The conversion between these equivalent data structures however remains implicit on paper, as they code in fact for the same free commutative algebra. In some other cases, implementation details are just ignored on paper, *e.g.*, when a proof involves both reasoning with Peano arithmetic and computing with large integers.

Example 1 (Proof-oriented vs. computation-oriented data structures). The standard library of the Coq interactive theorem prover [32] has two data structures

^{*} This work has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No.101001995).

for representing natural numbers. Type \mathbb{N} is the base-1 number system and the associated elimination principle \mathbb{N}_{ind} is the usual recurrence scheme:

```
Inductive N : Type := 0N : N | SN (n : N) : N.
N_ind : ∀ P : N → □, P 0N → (∀ n : N, P n → P (SN n)) → ∀ n : N, P n
```

On the other hand, type \mathbb{N} provides a binary representation `positive` of non-negative integers, as sequences of bits with a head 1, and is thus better suited for coding efficient arithmetic operations. The successor function $\mathbb{S}_N : \mathbb{N} \rightarrow \mathbb{N}$ is no longer a constructor of the type, but can be implemented as a program, via an auxiliary successor function \mathbb{S}_{pos} for type `positive`.

```
Inductive positive : Type :=
  xI : positive → positive | x0 : positive → positive | xH : positive.
Inductive N : Type := 0N : N | Npos : positive → N.
Fixpoint Spos (p : positive) : positive := match p with
  | xH ⇒ x0 xH | x0 p ⇒ xI p | xI p ⇒ x0 (Spos p) end.
Definition SN (n : N) := match n with
  | Npos p ⇒ Npos (Spos p) | _ ⇒ Npos xH end.
```

This successor function is useful to implement conversions $\uparrow_N : \mathbb{N} \rightarrow \mathbb{N}$ and $\downarrow_N : \mathbb{N} \rightarrow \mathbb{N}$ between the unary and binary representations. These conversion functions are in fact inverses of each other. The natural recurrence scheme on natural numbers thus *transfers* to type \mathbb{N} :

```
N_ind : ∀ P : N → □, P 0N → (∀ n : N, P n → P (SN n)) → ∀ n : N, P n
```

Incidentally, \mathbb{N}_{ind} can be proved from \mathbb{N}_{ind} by using only the fact that \downarrow_N is a left inverse of \uparrow_N , and the following compatibility lemmas:

$$\downarrow_N 0_N = 0_N \quad \text{and} \quad \forall n : \mathbb{N}, \quad \downarrow_N (\mathbb{S}_N n) = \mathbb{S}_N (\downarrow_N n)$$

Proof transfer issues are not tied to program verification. For instance, the formal study of summation and integration, in basic real analysis, provides a classic example of frustrating bureaucracy.

Example 2 (Extended domains). Given a sequence $(u_n)_{n \in \mathbb{N}}$ of non-negative real numbers, *i.e.*, a function $u : \mathbb{N} \rightarrow [0, +\infty[$, u is said to be *summable* when the sequence $(\sum_{k=0}^n u_k)_{n \in \mathbb{N}}$ has a finite limit, denoted $\sum u$. Now for two summable sequences u and v , it is easy to see that $u+v$, the sequence obtained by point-wise addition of u and v , is also a summable sequence, and that:

$$\sum (u + v) = \sum u + \sum v \tag{1}$$

As expression $\sum u$ only makes sense when u is a summable sequence, any algebraic operation “under the sum”, *e.g.*, rewriting $\sum (u + (v + w))$ into $\sum ((w + u) + v)$, *a priori* requires a proof of summability for every rewriting step. In a classical

setting, the standard approach rather assigns a default value to the case of an infinite sum, and introduces an extended domain $[0, +\infty]$. Algebraic operations on real numbers, like addition, are extended to the extra $+\infty$ case. Now for a sequence $u : \mathbb{N} \rightarrow [0, +\infty]$, the limit $\sum u$ is always defined, as increasing partial sums either converge to a finite limit, or diverge to $+\infty$. The road map is then to first prove that Equation 1 holds for *any* two sequences of *extended* non-negative numbers. The result is then *transferred* to the special case of summable sequences of non-negative numbers. Major libraries of formalized mathematics including Lean’s `mathlib` [1], Isabelle/HOL’s `Archive of Formal Proofs`, `coq-interval` [20] or Coq’s `mathcomp-analysis` [2], resort to such extended domains and transfer steps, notably for defining measure theory. Yet, as reported by expert users [18], the associated transfer bureaucracy is essentially done manually and thus significantly clutters formal developments in real and complex analysis, probabilities, etc.

Users of interactive theorem provers should be allowed to elude mundane arguments pertaining to proof transfer, as they would on paper, and spare themselves the related bureaucracy. Yet, they still need to convince the proof checker and thus have to provide explicit transfer proofs, albeit ideally automatically generated ones. The present work aims at providing a general method for implementing this nature of automation, for a diverse range of proof transfer problems.

In this paper, we focus on interactive theorem provers based on dependent type theory, such as Coq, Agda [24] or Lean [22]. These proof management systems are genuine functional programming languages, with full-spectrum dependent types, a context in which representation independence meta-theorems can be turned into concrete instruments for achieving program and proof transfer.

Seminal results on the contextual equivalence of distinct implementations of a same abstract interface were obtained for System F, using logical relations [21] and parametricity meta-theorems [26,35]. In the context of type theory, such meta-theorems can be turned into syntactic translations of the type theory of interest into itself, automating this way the generation of the statement and proof of parametricity properties for type families and for programs. Such syntactic relational models can accommodate dependent types [10], inductive types [9] and scale to the Calculus of Inductive Constructions, with an impredicative sort [19].

In particular, the *univalent parametricity* translation [30] leverages the univalence axiom [33] so as to transfer statements using established equivalences of types. This approach crucially removes the need for devising an explicit common interface for the types in relation. In presence of an internalized univalence axiom and of higher-inductive types, the *structure identity principle* provides internal representations of independence results, for more general relations between types than equivalences [5]. This last approach is thus particularly relevant in cubical type theory [12,34]. Indeed, a computational interpretation of the univalence axiom brings computational adequacy to otherwise possibly stuck terms, those resulting from a transfer involving an axiomatized univalence principle.

Yet taming the bureaucracy of proof transfer remains hard in practice for users of Coq, Lean or Agda. Examples 1 and 2 actually illustrate fundamental limitations of the existing approaches:

Univalence is overkill Both univalent parametricity and the structure identity principle can be used to derive the statement and the proof of the induction principle `N_ind` of Example 1, from the elimination scheme of type \mathbb{N} . But up to our knowledge, all the existing methods for automating this implication pull in the univalence principle in the proof, although it can be obtained by hand by very elementary means. This limitation is especially unsatisfactory for developers of libraries formalizing classical mathematics, and notably *Lean*’s `mathlib`. These libraries indeed typically assume a strong form of proof irrelevance, which is incompatible with univalence, and thus with univalent parametricity.

Equivalences are not enough, neither are quotients Univalent parametricity cannot help with Example 2, as type $[0, +\infty[$ is *not equivalent* to its extended version $[0, +\infty]$. In fact, we are not aware of any tool able to automate this proof transfer. In particular, the structure identity principle [5] would not apply as such.

Contributions In short, existing techniques for transferring results from one type to another, *e.g.*, from \mathbb{N} to \mathbb{N} or from extended real numbers to real numbers, are either not suitable for dependent types, or too coarse to track the exact amount of data needed in a given proof, and not more. This paper presents three contributions improving this unfortunate state of affairs:

- A parametricity framework *à la carte*, that generalizes the univalent parametricity translation [30], as well as refinements *à la* `CoqEAL` [14] and generalized rewriting [28]. Its pivotal ingredient is a variant of Altenkirch and Kaposi’s symmetrical presentation of type equivalence [3].
- A conservative subtyping extension of CC_ω [15], used to formulate an inference algorithm for the synthesis of parametricity proofs.
- The implementation of a new parametricity plugin for the `Coq` interactive theorem prover, using the `Coq-Elpi` [31] meta-language. This plugin rests on original formal proofs, conducted on top of the `HoTT` library [8], and is distributed with a collection of application examples.

Outline The rest of this paper is organized as follows. Section 2 introduces proof transfer and recalls the principle, strengths and weaknesses of the univalent parametricity translation. In Section 3, we present a new definition of type equivalence, motivating a hierarchy of structures for relations preserved by parametricity. Section 4 then presents variants of parametricity translations. In Section 5, we discuss a few examples of applications and we conclude in Section 6.

2 Strengths and limits of univalent parametricity

We first clarify the essence of proof transfer in dependent type theory (§ 2.1) and briefly recall a few concepts related to type equivalence and univalence (§ 2.2). We then review and discuss the limits of univalent parametricity (§ 2.3).

2.1 Proof transfer in type theory

We recall the syntax of the Calculus of Constructions, CC_ω , a λ -calculus with dependent function types and a predicative hierarchy of universes, denoted \square_i :

$$A, B, M, N ::= \square_i \mid x \mid M \ N \mid \lambda x : A. M \mid \Pi x : A. B$$

We omit the typing rules of the calculus, and refer the reader to standard references (*e.g.*, [25,23]). We also use the standard equality type, called propositional equality, as well as dependent pairs, denoted $\Sigma x : A. B$. We write $t \equiv u$ the definitional equality between two terms t and u . Interactive theorem provers like Coq, Agda and Lean are based on various extensions of this core, notably with inductive types or with an impredicative sort. When the universe level does not matter, we casually remove the annotation and use notation \square .

In this context, proof transfer from type T_1 to type T_2 roughly amounts to *synthesizing* a new type former $W : T_2 \rightarrow \square$, *i.e.*, a type parametric in some type T_2 , from an initial type former $V : T_1 \rightarrow \square$, *i.e.*, a type parametric in some type T_1 , so as to ensure that for some given relations $R_T : T_1 \rightarrow T_2 \rightarrow \square$ and $R_\square : \square \rightarrow \square \rightarrow \square$, there is a proof w that:

$$\Gamma \vdash w : \forall (t_1 : T_1)(t_2 : T_2), R_T \ t_1 \ t_2 \rightarrow R_\square (V \ t_1)(W \ t_2)$$

for a suitable context Γ . This setting generalizes as expected to k -ary type formers, and to more pairs of related types. In practice, relation R_\square is often a right-to-left arrow, *i.e.*, $R_\square \ A \ B \triangleq B \rightarrow A$, as in this case the proof w substantiates a proof step turning a goal clause $\Gamma \vdash V \ t_1$ into $\Gamma \vdash W \ t_2$.

Phrased as such, this synthesis problem is arguably quite loosely specified. Consider for instance the transfer problem discussed in Example 1. A first possible formalization involves the design of an appropriate common interface structure for types \mathbb{N} and \mathbb{N} , for instance by setting both T_1 and T_2 as $\Sigma N : \square. N \times (N \rightarrow N)$, and both V and W as: $\lambda X : T_1. \Pi P : X.1 \rightarrow \square. P \ X.2 \rightarrow (\Pi n : X.1. P \ n \rightarrow P \ (X.3 \ n)) \rightarrow \Pi n : X.1. P \ n$, where $X.i$ denotes the i -th item in the dependent tuple X . In this case, relation R_T may characterize isomorphic instances of the structure. Such instances of proof transfer are elegantly addressed in cubical type theories via internal representation independence results [5]. In the context of CC_ω , the hassle of devising explicit structures by hand has been termed the *anticipation* problem [30].

Another option is to consider two different types $T_1 \triangleq \mathbb{N} \times (\mathbb{N} \rightarrow \mathbb{N})$ and $T_2 \triangleq \mathbb{N} \times (\mathbb{N} \rightarrow \mathbb{N})$ and

$$\begin{aligned} V' &\triangleq \lambda X : T_1. \forall P : \mathbb{N} \rightarrow \square. P \ X.1 \rightarrow (\forall n : \mathbb{N}, P \ n \rightarrow P \ (X.2 \ n)) \rightarrow \forall n : \mathbb{N}, P \ n \\ W' &\triangleq \lambda X : T_2. \forall P : \mathbb{N} \rightarrow \square. P \ X.1 \rightarrow (\forall n : \mathbb{N}, P \ n \rightarrow P \ (X.2 \ n)) \rightarrow \forall n : \mathbb{N}, P \ n \end{aligned}$$

where one would typically expect R_T to be a type equivalence between T_1 and T_2 , so as to transport $(V' \ t_1)$ to $(W' \ t_2)$, along this equivalence.

Note that some solutions of given instances of proof transfer problems are in fact too trivial to be of interest. Consider for example the case of a *functional*

relation between T_2 and T_1 , with $R_T \ t_1 \ t_2$ defined as $t_1 = \phi \ t_2$, for some $\phi : T_2 \rightarrow T_1$. In this case, the composition $V \circ \phi$ is an obvious candidate for W , but is often uninformative. Indeed, this composition can only propagate structural arguments, blind to the additional mathematical proofs of program equivalences potentially available in the context. For instance, here is a useless variant of W' :

$$W'' \triangleq \lambda X : T_2. \ \forall P : \mathbb{N} \rightarrow \square. \ P \ (\uparrow_{\mathbb{N}} X.1) \rightarrow \\ (\forall n : \mathbb{N}, P \ n \rightarrow P \ (\uparrow_{\mathbb{N}} (X.2 \ (\downarrow_{\mathbb{N}} n)))) \rightarrow \forall n : \mathbb{N}, P \ n.$$

Automation devices dedicated to proof transfer thus typically consist of a meta-program which attempts to compute type former W and proof w by induction on the structure of V , by composing registered canonical pairs of related terms, and the corresponding proofs. These tools differ by the nature of relations they can accommodate, and by the class of type formers they are able to synthesize. For instance, *generalized rewriting* [28], which provides essential support to formalizations based on setoids [7], addresses the case of homogeneous (and reflexive) relations, *i.e.*, when T_1 and T_2 coincide. The CoqEAL library [14] provides another example of such transfer automation tool, geared towards *refinements*, typically from a proof-oriented data-structure to a computation-oriented one. It is thus specialized to heterogeneous, functional relations but restricted to closed, quantifier-free type formers. We now discuss the few transfer methods which can accommodate dependent types and heterogeneous relations.

2.2 Type equivalences, univalence

Let us first focus on the special case of types related by an *equivalence*, and start with a few standard definitions, notations and lemmas. Omitted details can be found in the usual references, like the Homotopy Type Theory book [33]. Two functions $f, g : A \rightarrow B$ are *point-wise equal*, denoted $f \doteq g$ when their values coincide on all arguments, that is $f \doteq g \triangleq \Pi a : A. f \ a = g \ a$. For any type A , id_A denotes $\lambda a : A. a$, the identity function on A , and we write id when the implicit type A is not ambiguous.

Definition 1 (Type isomorphism, type equivalence). *A function $f : A \rightarrow B$ is an isomorphism, denoted $IsIso(f)$, if there exists a function $g : B \rightarrow A$ which satisfies the section and retraction properties, *i.e.*, g is respectively a point-wise left and right inverse of f . A function f is an equivalence, denoted $IsEquiv(f)$, when it moreover enjoys a coherence property, relating the proofs of the section and retraction properties and ensuring that $IsEquiv(f)$ is proof-irrelevant.*

Types A and B are equivalent, denoted $A \simeq B$, when there is an equivalence $f : A \rightarrow B$:

$$A \simeq B \triangleq \Sigma f : A \rightarrow B. IsEquiv(f)$$

Lemma 1. *Any isomorphism $f : A \rightarrow B$ is also an equivalence.*

The data of an equivalence $e : A \simeq B$ thus include two *transport functions*, denoted respectively $\uparrow_e : A \rightarrow B$ and $\downarrow_e : B \rightarrow A$. They can be used for proof

transfer from A to B , using \uparrow_e at covariant occurrences, and \downarrow_e at contravariant ones. The *univalence principle* asserts that equivalent types are interchangeable, in the sense that all universes are univalent.

Definition 2 (Univalent universe). *A universe \mathcal{U} is univalent if for any two types A and B in \mathcal{U} , the canonical map $A = B \rightarrow A \simeq B$ is an equivalence.*

In variants of CC_ω , the *univalence axiom* has no explicit computational content: it just postulates that all universes \square_i are univalent, as for instance in the **HoTT** library for the **Coq** interactive theorem prover [8]. Some more recent variants of dependent type theory [12,4] feature a built-in computational univalence principle. They are used to implement experimental interactive theorem provers, such as **Cubical Agda** [34]. In both cases, the univalence principle provides a powerful proof transfer principle from \square to \square , as for any two types A and B such that $A \simeq B$, and any $P : \square \rightarrow \square$, we can obtain that $P A \simeq P B$ as a direct corollary of univalence. Concretely, $P B$ is obtained from $P A$ by appropriately allocating the transfer functions provided by the equivalence data, a transfer process typically useful in the context of proof engineering [27].

Going back to our example from § 2.1, transferring along an equivalence $\mathbb{N} \simeq \mathbb{N}$ thus produces W'' from V' . Assuming univalence, one may achieve the more informative transport from V' to W' , using a method called *univalent parametricity* [30], which we discuss in the next section.

2.3 Parametricity translations

Univalent parametricity strengthens the transfer principle provided by the univalence axiom by combining it with parametricity. In CC_ω , the essence of parametricity, which is to devise a relational interpretation of types, can be turned into an actual syntactic translation, as relations can themselves be modeled as λ -terms in CC_ω . The seminal work of Bernardy, Lasson *et al.* [10,19,9] combine in what we refer to as the *raw parametricity translation*, which essentially defines inductively a logical relation $\llbracket T \rrbracket$ for any type T , as described on Figure 1. This presentation uses the standard convention that t' is the term obtained from a term t by replacing every variable x in t with a fresh variable x' . A variable x is translated into a variable x_R , where x_R is a fresh name. Parametricity follows from the associated fundamental theorem, also called abstraction theorem [26]:

Theorem 1. *If $\Gamma \vdash t : T$ then the following hold: $\llbracket \Gamma \rrbracket \vdash t : T$, $\llbracket \Gamma \rrbracket \vdash t' : T'$ and $\llbracket \Gamma \rrbracket \vdash \llbracket t \rrbracket : \llbracket T \rrbracket t t'$.*

Proof. By structural induction on the typing judgment, see for instance [19].

A key, albeit mundane ingredient of Theorem 1 is the fact that the rules of Figure 1 ensure that:

$$\vdash \llbracket \square_i \rrbracket : \llbracket \square_{i+1} \rrbracket \square_i \square_i \quad (9)$$

This translation precisely generates the statements expected from a parametric type family or program. For instance, the translation of a Π -type, given by

– Context translation:

$$\llbracket \langle \rangle \rrbracket = \langle \rangle \quad (2)$$

$$\llbracket \Gamma, x : A \rrbracket = \llbracket \Gamma \rrbracket, x : A, x' : A', x_R : \llbracket A \rrbracket x x' \quad (3)$$

– Term translation:

$$\llbracket \square_i \rrbracket = \lambda A A'. A \rightarrow A' \rightarrow \square_i \quad (4)$$

$$\llbracket x \rrbracket = x_R \quad (5)$$

$$\llbracket A B \rrbracket = \llbracket A \rrbracket B B' \llbracket B \rrbracket \quad (6)$$

$$\llbracket \lambda x : A. t \rrbracket = \lambda(x : A)(x' : A')(x_R : \llbracket A \rrbracket x x'). \llbracket t \rrbracket \quad (7)$$

$$\llbracket \Pi x : A. B \rrbracket = \lambda f f'. \Pi(x : A)(x' : A')(x_R : \llbracket A \rrbracket x x'). \llbracket B \rrbracket(f x)(f' x') \quad (8)$$

Fig. 1: Raw parametricity translation for CC_ω .

Equation 8, is a type of relations on functions that relate those producing related outputs from related inputs. Concrete implementations of this translation are available [19,31]; they generate and prove parametricity properties for type families or for constants, improved induction schemes, etc.

Univalent parametricity follows from the observation that the abstraction theorem still holds when restricting to relations that are in fact (heterogeneous) equivalences. This however requires some care in the translation of universes:

$$\begin{aligned} \llbracket \square_i \rrbracket A B &\triangleq \Sigma(R : A \rightarrow B \rightarrow \square_i)(e : A \simeq B). \\ &\quad \Pi(a : A)(b : B). R a b \simeq (a =_{\downarrow_e} b) \end{aligned} \quad (10)$$

where $\llbracket \cdot \rrbracket$ now refers to the *univalent* parametricity translation, replacing the notation introduced for the raw variant. For any two types A and B , $\llbracket \square_i \rrbracket A B$ packages a relation R and an equivalence e such that R is equivalent to the functional relation associated with \downarrow_e . Crucially, assuming univalence, $\llbracket \square_i \rrbracket$ is equivalent to type equivalence, that is, for any two types A and B :

$$\llbracket \square_i \rrbracket A B \simeq (A \simeq B).$$

This observation is actually an instance of a more general technique available for constructing syntactic models of type theory [11], based on attaching extra intensional specifications to negative type constructors. In these models, a standard way to recover the abstraction theorem consists of refining the translation into two variants, for any term $T : \square_i$, that is also a type. The translation of such a T as a *term*, denoted $\llbracket T \rrbracket$, is a dependent pair, which equips a relation with the additional data prescribed by the interpretation $\llbracket \square_i \rrbracket$ of the universe. The translation $\llbracket T \rrbracket$ of T as a *type* is the relation itself, that is, the projection of the dependent pair $\llbracket T \rrbracket$ onto its first component, denoted $\text{rel}(\llbracket T \rrbracket)$. We refer to the original article [30, Figure 4] for a complete description of the translation.

We now state the abstraction theorem of the univalent parametricity translation [30], where \vdash_u denotes a typing judgment of CC_ω assuming univalence:

Theorem 2. *If $\Gamma \vdash t : T$ then $\llbracket \Gamma \rrbracket \vdash_u [t] : \llbracket T \rrbracket t t'$.*

Note that proving the abstraction theorem 2 involves in particular proving that:

$$\vdash_u [\Box_i] : \llbracket \Box_{i+1} \rrbracket \Box_i \Box_i \quad \text{and} \quad \text{rel}([\Box_i]) \equiv \llbracket \Box_i \rrbracket. \quad (11)$$

The definition of relation $[\Box_i]$ relies on the univalence principle in a crucial way, in order to prove that the relation in the universe is equivalent to equality on the universe, *i.e.*, to prove that:

$$\vdash_u \Pi A B : \Box_i. \llbracket \Box_i \rrbracket A B \simeq (A = B).$$

Importantly, this univalent parametricity translation can be seamlessly extended so as to also make use of a global context of user-defined equivalences.

Yet because of the interpretation of universes given by Equation 10, univalent parametricity can only automate proof transfer based on *type equivalences*. This is too strong a requirement in many cases, *e.g.*, to deduce properties of natural numbers from that of integers, or more generally for refinement relations. Even in the case of equivalent types, this restriction may be problematic, as Equation 11 may incur unnecessary dependencies on the univalence axiom, as in Example 1.

3 Type equivalence in kit

In this section, we propose (§ 3.1) an equivalent, modular presentation of type equivalence, phrased as a nested sigma type. Then (§ 3.2), we carve a hierarchy of structures on relations out of this dependent tuple, selectively picking pieces. Last, we revisit (§ 3.3) parametricity translations through the lens of this finer grained analysis of the relational interpretations of types.

3.1 Disassembling type equivalence

Let us first observe that the Definition 1, of type equivalence, is quite asymmetrical, although this fact is somehow swept under the rug by the infix $A \simeq B$ notation. First, the data of an equivalence $e : A \simeq B$ privileges the left-to-right direction, as \uparrow_e is directly accessible from e as its first projection, while accessing the right-to-left transport requires an additional projection. Second, the statement of the coherence property, which we eluded in Definition 1, is actually:

$$\Pi a : A. \text{ap}_{\uparrow_e}(s \ a) = r \circ \downarrow_e$$

where $\text{ap}_f(t)$ is the term $f \ u = f \ v$, for any identity proof $t : u = v$. This statement uses proofs s and r , respectively of the section and retraction properties of e , but not in a symmetrical way, although swapping them leads to an equivalent definition. This entanglement prevents tracing the respective roles of each direction of transport, left-to-right or right-to-left, during the course of a given univalent parametricity translation. Exercise 4.2 in the HoTT book [33] however suggests a symmetrical definition of type equivalence, via functional relations.

Definition 3. A relation $R : A \rightarrow B \rightarrow \square_i$, is functional when:

$$\Pi a : A. \text{IsContr}(\Sigma b : B. R \ a \ b)$$

where for any type T , $\text{IsContr}(T)$ is the standard contractibility predicate $\Sigma t : T. \Pi t' : T. t = t'$. This property is denoted $\text{IsFun}(R)$.

We can now obtain an equivalent but symmetrical characterization of type equivalence, as a functional relation whose symmetrization is also functional.

Lemma 2. For any types $A, B : \square$, type $A \simeq B$ is equivalent to:

$$\Sigma R : A \rightarrow B \rightarrow \square. \text{IsFun}(R) \times \text{IsFun}(R^{-1})$$

where $R^{-1} : B \rightarrow A \rightarrow \square$ just swaps the arguments of relation $R : A \rightarrow B \rightarrow \square$.

We sketch below a proof of this result, left as an exercise in [33]. The essential argument is the following characterization of functional relations:

Lemma 3. The type of functions is equivalent to the type of functional relations; i.e., for any types $A, B : \square$, we have $(A \rightarrow B) \simeq \Sigma R : A \rightarrow B \rightarrow \square. \text{IsFun}(R)$.

Proof. The proof goes by chaining the following equivalences:

$$\begin{aligned} (\Sigma R : A \rightarrow B \rightarrow \square. \text{IsFun}(R)) &\simeq (A \rightarrow \Sigma P : B \rightarrow \square. \text{IsContr}(\Sigma b : B. P \ b)) \\ &\simeq (A \rightarrow B) \end{aligned}$$

Proof (of Lemma 2). The proof goes by chaining the following equivalences, where the type of f is always $A \rightarrow B$ and the type of R is $A \rightarrow B \rightarrow \square$:

$$\begin{aligned} (A \simeq B) &\simeq \Sigma f : A \rightarrow B. \text{IsEquiv}(f) && \text{by definition of } (A \simeq B) \\ &\simeq \Sigma f. \Pi b : B. \text{IsContr}(\Sigma a. f \ a = b) && \text{standard result in HoTT} \\ &\simeq \Sigma f. \text{IsFun}(\lambda(b : B)(a : A). f \ a = b) && \text{by definition of } \text{IsFun}(\cdot) \\ &\simeq \Sigma (\varphi : \Sigma R. \text{IsFun}(R)). \text{IsFun}(\pi_1(\varphi)^{-1}) && \text{by Lemma 3} \\ &\simeq \Sigma R. \text{IsFun}(R) \times \text{IsFun}(R^{-1}) && \text{by associativity of } \Sigma. \end{aligned}$$

However, the definition of type equivalence provided by Lemma 2 does not expose explicitly the two transfer functions in its data, although this computational content can be extracted via first projections of contractibility proofs. In fact, it is possible to devise a definition of type equivalence which directly provides the two transport functions in its data, while remaining symmetrical. This variant follows from an alternative characterization of functional relations.

Definition 4. For any types $A, B : \square$, a relation $R : A \rightarrow B \rightarrow \square$, is a univalent map, denoted $\text{IsUmap}(R)$ when there exists a function $m : A \rightarrow B$ together with:

$$\begin{aligned} g_1 : \Pi(a : A)(b : B). m \ a = b \rightarrow R \ a \ b \\ \text{and } g_2 : \Pi(a : A)(b : B). R \ a \ b \rightarrow m \ a = b \\ \text{such that } \Pi(a : A)(b : B). (g_1 \ a \ b) \circ (g_2 \ a \ b) \doteq \text{id}. \end{aligned}$$

Now comes the crux lemma of this section.

Lemma 4. *For any types $A, B : \square$ and any relation $R : A \rightarrow B \rightarrow \square$*

$$IsFun(R) \simeq IsUmap(R).$$

Proof. The proof goes by rewording the left hand side, in the following way:

$$\begin{aligned} & \Pi x. IsContr(R\ x) \\ & \simeq \Pi x. \Sigma(r : \Sigma y. R\ x\ y). \Pi(p : \Sigma y. R\ x\ y). r = p \\ & \simeq \Pi x. \Sigma y. \Sigma(r : R\ x\ y). \Pi(p : \Sigma y. R\ x\ y). (y, r) = p \\ & \simeq \Sigma f. \Pi x. \Sigma(r : R\ x\ (f\ x)). \Pi(p : \Sigma y. R\ x\ y). (f\ x, r) = p \\ & \simeq \Sigma f. \Sigma(r : \Pi x. R\ x\ (f\ x)). \Pi x. \Pi(p : \Sigma y. R\ x\ y). (f\ x, r\ x) = p \\ & \simeq \Sigma f. \Sigma r. \Pi x. \Pi y. \Pi(p : R\ x\ y). (f\ x, r\ x) = (y, p) \\ & \simeq \Sigma f. \Sigma r. \Pi x. \Pi y. \Pi(p : R\ x\ y). \Sigma(e : f\ x = y). r\ x =_e p \\ & \simeq \Sigma f. \Sigma r. \Sigma(e : \Pi x. \Pi y. R\ x\ y \rightarrow f\ x = y). \Pi x. \Pi y. \Pi p. (r\ x) =_{e\ x\ y\ p} p \end{aligned}$$

After a suitable reorganization of the sigma types we are left to show that

$$\begin{aligned} & \Sigma(r : \Pi x. \Pi y. f\ x = y \rightarrow R\ x\ y). (e\ x\ y) \circ (r\ x\ y) \doteq id \\ & \simeq \Sigma(r : \Pi x. R\ x\ (f\ x)). \Pi x. \Pi y. \Pi p. r\ x =_{e\ x\ y\ p} p \end{aligned}$$

which proof we do not detail, referring the reader to the supplementary material.

As a direct corollary, we obtain a novel characterization of type equivalence:

Theorem 3. *For any types $A, B : \square_i$, we have:*

$$(A \simeq B) \simeq \boxdot^\top A\ B$$

where the relation $\boxdot^\top A\ B$ is defined as:

$$\Sigma R : A \rightarrow B \rightarrow \square_i. IsUmap(R) \times IsUmap(R^{-1})$$

The collection of data packed in a term of type $\boxdot^\top A\ B$ is now symmetrical, as the right-to-left direction of the equivalence based on univalent maps can be obtained from the left-to-right by flipping the relation and swapping the two functionality proofs. If the η -rule for records is verified, symmetry is even *definitionally* involutive.

3.2 Reassembling type equivalence

Definition 4 of univalent maps and the resulting rephrasing of type equivalence suggest introducing a hierarchy of structures for heterogeneous relations, which explains how close a given relation is to type equivalence. In turn, this distance is described in terms of structure available respectively on the left-to-right and right-to-left transport functions.

Definition 5. For $n, k \in \{0, 1, 2_a, 2_b, 3, 4\}$, and $\alpha = (n, k)$, relation $\boxdot^\alpha : \Box \rightarrow \Box \rightarrow \Box$, is defined as:

$$\boxdot^\alpha \triangleq \lambda(A \ B : \Box). \Sigma(R : A \rightarrow B \rightarrow \Box). \text{Class}_\alpha R$$

where the map class $\text{Class}_\alpha R$ itself unfolds to a pair type $(M_n R) \times (M_k R^{-1})$, with M_i defined as:⁵

$$M_0 R \triangleq .$$

$$M_1 R \triangleq (A \rightarrow B)$$

$$M_{2_a} R \triangleq \Sigma m : A \rightarrow B. G_{2_a} m R \quad \text{with } G_{2_a} m R \triangleq \Pi a b. m a = b \rightarrow R a b$$

$$M_{2_b} R \triangleq \Sigma m : A \rightarrow B. G_{2_b} m R \quad \text{with } G_{2_b} m R \triangleq \Pi a b. R a b \rightarrow m a = b$$

$$M_3 R \triangleq \Sigma m : A \rightarrow B. (G_{2_a} m R) \times (G_{2_b} m R)$$

$$M_4 R \triangleq \Sigma m : A \rightarrow B. \Sigma(g_1 : G_{2_a} m R). \Sigma(g_2 : G_{2_b} m R). \Pi a b.$$

$$(g_1 a b) \circ (g_2 a b) \doteq \text{id}$$

For any types A and B , and any $r : \boxdot^\alpha A B$ we use notations $\text{rel}(r)$, $\text{map}(r)$ and $\text{comap}(r)$ to refer respectively to the relation, map of type $A \rightarrow B$, map of type $B \rightarrow A$, included in the data of r , for a suitable α .

Definition 6. We denote \mathcal{A} the set $\{0, 1, 2_a, 2_b, 3, 4\}^2$, used to index map classes in Definition 5. This set is partially ordered for the product order defined from the partial order $0 < 1 < 2_* < 3 < 4$ for 2_* either 2_a or 2_b , and with 2_a and 2_b being incomparable.

Remark 1. Relation $\boxdot^{(4,4)}$ of Definition 5 coincides with the relation \boxdot^\top introduced in Theorem 3. Similarly, we denote \boxdot^\perp the relation $\boxdot^{(0,0)}$.

Remark 2. Definition 5 is associated with the following dictionary. For r of type:

- $\boxdot^{(1,0)} A B$, $\text{map}(r)$ is an arbitrary function $f : A \rightarrow B$;
- $\boxdot^{(4,0)} A B$, $\text{rel}(r)$ is a univalent map, in the sense of Definition 4;
- $\boxdot^{(4,2_a)} A B$, $\text{rel}(r)$ is the graph of a retraction (*i.e.*, a surjective univalent map with an explicit partial left inverse) of type $A \rightarrow B$;
- $\boxdot^{(4,2_b)} A B$, $\text{rel}(r)$ is the graph of a section (*i.e.*, an injective univalent map with explicit partial right inverse) of type $A \rightarrow B$;
- $\boxdot^{(4,4)}$, r is an equivalence between A and B ;
- $\boxdot^{(3,3)}$, r is an isomorphism between A and B .

Observe that $\boxdot^{(n,m)} A B$ coincides, up to equivalence, with $\boxdot^{(m,n)} B A$. Other classes, while not corresponding to a meaningful mathematical definition, may arise in concrete runs of proof transfer: see also Section 4 for explicit examples.

The corresponding lattice to the collection of M_n is implemented as a hierarchy of dependent tuples, more precisely, of record types.

⁵ For the sake of readability, we omit implicit arguments, *e.g.*, although M_i has type $\lambda(T_1 T_2 : \Box). (T_1 \rightarrow T_2 \rightarrow \Box) \rightarrow \Box$, we write $M_n R$ for $(M_n A B R)$.

3.3 Populating the hierarchy of relations

We shall now revisit the parametricity translations of Section 2.3. In particular, combining Theorem 3 with Equation 11, crux of the abstraction theorem for univalent parametricity, ensures the existence of a term p_{\square_i} such that:

$$\vdash_u p_{\square_i} : \boxdot_{i+1}^\top \square_i \square_i \quad \text{and} \quad \text{rel}(p_{\square_i}) \simeq \boxdot_i^\top.$$

Otherwise said, relation $\boxdot^\top : \square \rightarrow \square \rightarrow \square$ can be endowed with a \boxdot^\top structure, assuming univalence. Similarly, Equation 9, for the raw parametricity translation, can be read as the fact that relation \boxdot^\perp on universes can be endowed with a $\boxdot^\perp \square \square$ structure.

Now the hierarchy of structures introduced by Definition 5 enables a finer grained analysis of the possible relational interpretations of universes. Not only would this put the raw and univalent parametricity translations under the same hood, but it would also allow for generalizing parametricity to a larger class of relations. For this purpose, we generalize the previous observation, on the key ingredient for translating universes: for each $\alpha \in \mathcal{A}$, relation $\boxdot^\alpha : \square \rightarrow \square \rightarrow \square$ may be endowed with several structures from the lattice, and we need to study which ones, depending on α . Otherwise said, we need to identify the pairs $(\alpha, \beta) \in \mathcal{A}^2$ for which it is possible to construct a term $p_{\square}^{\alpha, \beta}$ such that:

$$\vdash_u p_{\square}^{\alpha, \beta} : \boxdot^\beta \square \square \quad \text{and} \quad \text{rel}(p_{\square}^{\alpha, \beta}) \equiv \boxdot^\alpha \quad (12)$$

Note that we aim here at a definitional equality between $\text{rel}(p_{\square}^{\alpha, \beta})$ and \boxdot^α , rather than at an equivalence. It is easy to see that a term $p_{\square}^{\alpha, \perp}$ exists for any $\alpha \in \mathcal{A}$, as \boxdot^\perp requires no structure on the relation. On the other hand, it is not possible to construct a term $p_{\square}^{\perp, \top}$, i.e., to turn an arbitrary relation into a type equivalence.

Definition 7. We denote \mathcal{D}_{\square} the following subset of \mathcal{A}^2 :

$$\mathcal{D}_{\square} = \{(\alpha, \beta) \in \mathcal{A}^2 \mid \alpha = \top \vee \beta \in \{0, 1, 2_a\}^2\}$$

The supplementary material constructs terms $p_{\square}^{\alpha, \beta}$ for every pair $(\alpha, \beta) \in \mathcal{D}_{\square}$, using a meta-program to generate them from a minimal collection of manual definitions. In particular, assuming univalence, it is possible to construct a term $p_{\square}^{\top, \top}$, which can be seen as an analogue of the translation $[\square]$ of univalent parametricity. More generally, the provided terms $p_{\square}^{\alpha, \beta}$ depend on univalence if and only if $\beta \notin \{0, 1, 2_a\}^2$.

The next natural question concerns the possible structures \boxdot^γ endowing the relational interpretation of a product type $\Pi x : A. B$, given relational interpretation for types A and B respectively equipped with structures \boxdot^α and \boxdot^β .

Otherwise said, we need to identify the triples $(\alpha, \beta, \gamma) \in \mathcal{A}^3$ for which it is possible to construct a term p_H^{γ} such that the following statements both hold:

$$\frac{\Gamma \vdash A_R : \boxdot^\alpha A \ A' \quad \Gamma, x : A, x' : A', x_R : A_R x \ x' \vdash B_R : \boxdot^\beta B \ B'}{\Gamma \vdash p_H^{\gamma} A_R B_R : \boxdot^\gamma (\Pi x : A. B) (\Pi x' : A'. B')}$$

$$\text{rel}(p_{\Pi}^{\gamma} A_R B_R) \equiv \lambda f. \lambda f'. \Pi(x : A)(x' : A')(x_R : \text{rel}(A_R) x x'). \text{rel}(B_R) (fx) (f'x')$$

The corresponding collection of triples can actually be described as a function $\mathcal{D}_{\Pi} : \mathcal{A} \rightarrow \mathcal{A}^2$, such that $\mathcal{D}_{\Pi}(\gamma) = (\alpha, \beta)$ provides the *minimal* requirements on the structures associated with A and B , with respect to the partial order on \mathcal{A}^2 . The supplementary material provides a corresponding collection of terms p_{Π}^{γ} for each $\gamma \in \mathcal{A}$, as well as all the associated weakenings. Once again, these definitions are generated by a meta-program. Observe in particular that by symmetry, $p_{\Pi}^{(m,n)}$ can be obtained from $p_{\Pi}^{(m,0)}$ and $p_{\Pi}^{(n,0)}$ by swapping the latter and glueing it to the former. Therefore, the values of p_{Π}^{γ} and $\mathcal{D}_{\Pi}(\gamma)$ are completely determined by those of $p_{\Pi}^{(m,0)}$ and $\mathcal{D}_{\Pi}(m, 0)$. In particular, for any $(m, n) \in \mathcal{A}$:

$$\mathcal{D}_{\Pi}(m, n) = ((m_A, n_A), (m_B, n_B))$$

where $m_A, n_A, m_B, n_B \in \mathcal{A}$ are such that $\mathcal{D}_{\Pi}(m, 0) = ((0, n_A), (m_B, 0))$ and $\mathcal{D}_{\Pi}(n, 0) = ((0, m_A), (n_B, 0))$. We sum up in Figure 2 the values of $\mathcal{D}_{\Pi}(m, 0)$.

| m | $\mathcal{D}_{\Pi}(m, 0)_1$ | $\mathcal{D}_{\Pi}(m, 0)_2$ | m | $\mathcal{D}_{\rightarrow}(m, 0)_1$ | $\mathcal{D}_{\rightarrow}(m, 0)_2$ |
|----------------|-----------------------------|-----------------------------|----------------|-------------------------------------|-------------------------------------|
| 0 | (0, 0) | (0, 0) | 0 | (0, 0) | (0, 0) |
| 1 | (0, 2 _a) | (1, 0) | 1 | (0, 1) | (1, 0) |
| 2 _a | (0, 4) | (2 _a , 0) | 2 _a | (0, 2 _b) | (2 _a , 0) |
| 2 _b | (0, 2 _a) | (2 _b , 0) | 2 _b | (0, 2 _a) | (2 _b , 0) |
| 3 | (0, 4) | (3, 0) | 3 | (0, 3) | (3, 0) |
| 4 | (0, 4) | (4, 0) | 4 | (0, 4) | (4, 0) |

Fig. 2: Minimal dependencies for product and arrow types

Note that in the case of a non-dependent product, constructing p_{\rightarrow}^{γ} requires less structure on the domain A of an arrow type $A \rightarrow B$, which motivates the introduction of function $\mathcal{D}_{\rightarrow}(\gamma)$. Using the combinator for dependent products to interpret an arrow type, albeit correct, potentially pulls in unnecessary structure (and axiom) requirements. The supplementary material includes a construction of terms p_{\rightarrow}^{γ} for any $\gamma \in \mathcal{A}$.

The two tables in Figure 2 show how requirements on levels stay the same on the right hand side of both Π and \rightarrow , stay the same up to symmetries (exchange of variance and of 2_a and 2_b) on the left hand side of a \rightarrow and increase on the left hand side of a Π . This elegant arithmetic justifies our hierarchy of relations.

4 A calculus for proof transfer

This section introduces TROCQ, a framework for proof transfer designed as a generalization of parametricity translations, so as to allow for interpreting types as instances of the structures introduced in Section 3.2. We adopt a sequent style presentation, which fits closely the type system of CC_{ω} , while explaining in a consistent way the transformations of terms and contexts. This choice of presentation departs from the standard literature about parametricity in pure type systems. Yet, it brings the presentation closer to actual implementations,

whose necessary management of parametricity contexts is swept under the rug by notational conventions (*e.g.*, the primes of Section 2.3).

For this purpose, we successively introduce four calculi, of increasing sophistication. We start (§ 4.1) with introducing this sequent style presentation by rephrasing the raw parametricity translation, and the univalent parametricity one (§ 4.2). We then introduce CC_ω^+ (§ 4.3), a Calculus of Constructions with annotations on sorts and subtyping, before defining (§ 4.4) the TROCQ calculus.

4.1 Raw parametricity sequents

We introduce *parametricity contexts*, under the form of a list of triples packaging two variables x and x' together with a third one x_R . The latter x_R is a *witness* (a proof) that x and x' are related:

$$\Xi ::= \varepsilon \mid \Xi, x \sim x' \because x_R$$

We write $(x, x', x_R) \in \Xi$ when $\Xi = \Xi', x \sim x' \because x_R, \Xi''$ for some Ξ' and Ξ'' .

We denote $\text{Var}(\Xi)$ the sequence of variables related in a parametricity context Ξ , with multiplicities:

$$\text{Var}(\varepsilon) = \varepsilon \quad \text{Var}(\Xi, x \sim x' \because x_R) = \text{Var}(\Xi), x, x', x_R$$

A parametricity context Ξ is *well-formed*, written $\Xi \vdash$, if the sequence $\text{Var}(\Xi)$ is duplicate-free. In this case, we use the notation $\Xi(x) = (x', x_R)$ as a synonym of $(x, x', x_R) \in \Xi$.

A *parametricity judgment* relates a parametricity context Ξ and three terms M, M', M_R of CC_ω . Parametricity judgments, denoted as:

$$\Xi \vdash M \sim M' \because M_R,$$

are defined by rules of Figure 3 and read *in context Ξ , term M translates to the term M' , because M_R* .

Lemma 5. *The relation associating a term M with pairs (M', M_R) such that $\Xi \vdash M \sim M' \because M_R$ holds, with Ξ a well-formed parametricity context, is functional. More precisely, for any well-formed parametricity context Ξ :*

$$\begin{aligned} \forall M, M', N', M_R, N_R, \quad \Xi \vdash M \sim M' \because M_R \quad \wedge \quad \Xi \vdash M \sim N' \because N_R \\ \implies (M', M_R) = (N', N_R) \end{aligned}$$

Proof. Immediate by induction on the syntax of M .

This presentation of parametricity thus provides an alternative definition of translation $\llbracket \cdot \rrbracket$ from Figure 1, and accounts for the prime-based notational convention used in the latter.

$$\begin{array}{c}
\frac{}{\Xi \vdash \Box_i \sim \Box_i \quad \cdot \quad \lambda(A B : \Box_i). A \rightarrow B \rightarrow \Box_i} \text{(PARAMSORT)} \\
\\
\frac{(x, x', x_R) \in \Xi \quad \Xi \vdash}{\Xi \vdash x \sim x' \quad \cdot \quad x_R} \text{(PARAMVAR)} \\
\\
\frac{\Xi \vdash M \sim M' \quad \cdot \quad M_R \quad \Xi \vdash N \sim N' \quad \cdot \quad N_R}{\Xi \vdash M N \sim M' N' \quad \cdot \quad M_R N N' N_R} \text{(PARAMAPP)} \\
\\
\frac{\Xi, x \sim x' \quad \cdot \quad x_R \vdash M \sim M' \quad \cdot \quad M_R}{\Xi \vdash \lambda x : A. M \sim \lambda x' : A'. M' \quad \cdot \quad \lambda x x' x_R. M_R} \text{(PARAMLAM)} \\
\\
\frac{x, x' \notin \text{Var}(\Xi) \quad \Xi \vdash A \sim A' \quad \cdot \quad A_R \quad \Xi, x \sim x' \quad \cdot \quad x_R \vdash B \sim B' \quad \cdot \quad B_R}{\Xi \vdash \Pi x : A. B \sim \Pi x' : A'. B' \quad \cdot \quad \lambda f g. \Pi x x' x_R. B_R (f x) (g x')} \text{(PARAMPI)}
\end{array}$$

Fig. 3: PARAM: sequent-style binary parametricity translation

Definition 8. A parametricity context Ξ is admissible for a well-formed typing context Γ , denoted $\Gamma \triangleright \Xi$, when Ξ and Γ are well-formed as a parametricity context and Γ provides coherent type annotations for all terms in Ξ , that is, for any variables x, x', x_R such that $\Xi(x) = (x', x_R)$, and for any terms A' and A_R :

$$\Xi \vdash \Gamma(x) \sim A' \quad \cdot \quad A_R \quad \implies \quad \Gamma(x') = A' \wedge \Gamma(x_R) \equiv A_R \ x \ x'$$

We can now state and prove an abstraction theorem:

Theorem 4 (Abstraction theorem).

$$\frac{\Gamma \triangleright \Xi \quad \Gamma \vdash M : A \quad \Xi \vdash M \sim M' \quad \cdot \quad M_R \quad \Xi \vdash A \sim A' \quad \cdot \quad A_R}{\Gamma \vdash M' : A' \quad \text{and} \quad \Gamma \vdash M_R : A_R \ M \ M'}$$

Proof. By induction on the derivation of $\Xi \vdash M \sim M' \quad \cdot \quad M_R$.

4.2 Univalent parametricity sequents

We now propose in Figure 4 a rephrased version of the univalent parametricity translation [30], using the same sequent style and replacing the translation of universes with the equivalent relation \Box^\top . Parametricity judgments are denoted:

$$\Xi \vdash_u M \sim M' \quad \cdot \quad M_R$$

where Ξ is a parametricity context and M, M' , and M_R are terms of CC_ω . The u index is a reminder that typing judgments $\Gamma \vdash_u M : A$ involved in the associated abstraction theorem assume the univalence axiom.

We can now rephrase the abstraction theorem for univalent parametricity.

$$\begin{array}{c}
\frac{}{\Xi \vdash_u \Box_i \sim \Box_i \therefore p_{\Box_i}^{\top, \top}} (\text{UPARAMSORT}) \quad \frac{(x, x', x_R) \in \Xi \quad \Xi \vdash}{\Xi \vdash_u x \sim x' \therefore x_R} (\text{UPARAMVAR}) \\
\\
\frac{\Xi \vdash_u M \sim M' \therefore M_R \quad \Xi \vdash_u N \sim N' \therefore N_R}{\Xi \vdash_u M N \sim M' N' \therefore M_R N N' N_R} (\text{UPARAMAPP}) \\
\\
\frac{\Xi \vdash_u A \sim A' \therefore A_R \quad \Xi, x \sim x' \therefore x_R \vdash_u M \sim M' \therefore M_R}{\Xi \vdash_u \lambda x : A. M \sim \lambda x' : A'. M' \therefore \lambda x x' x_R. M_R} (\text{UPARAMLAM}) \\
\\
\frac{\Xi \vdash_u A \sim A' \therefore A_R \quad \Xi, x \sim x' \therefore x_R \vdash_u B \sim B' \therefore B_R}{\Xi \vdash_u \Pi x : A. B \sim \Pi x' : A'. B' \therefore p_{\Pi}^{\top} A_R B_R} (\text{UPARAMPI})
\end{array}$$

Fig. 4: UPARAM: univalent parametricity rules

Theorem 5 (Univalent abstraction theorem).

$$\frac{\Gamma \triangleright \Xi \quad \Gamma \vdash M : A \quad \Xi \vdash_u M \sim M' \therefore M_R \quad \Xi \vdash_u A \sim A' \therefore A_R}{\Gamma \vdash M' : A' \quad \text{and} \quad \Xi \vdash_u M_R : \text{rel}(A_R) M M'}$$

Proof. By induction on the derivation of $\Xi \vdash_u M \sim M' \therefore M_R$.

Remark 3. In Theorem 5, $\text{rel}(A_R)$ is a term of type $A \rightarrow A' \rightarrow \Box$. Indeed:

$$\frac{\Gamma \vdash A : \Box_i \quad \Xi \vdash_u A \sim A' \therefore A_R \quad \Gamma \triangleright \Xi}{\Gamma \vdash_u A_R : \text{rel}(p_{\Box_i}^{\top, \top}) A A'}$$

entails A_R has type

$$\begin{aligned}
\text{rel}(p_{\Box_i}^{\top, \top}) A A' &\equiv \boxtimes^{\top} A A' \\
&\equiv \Sigma R : A \rightarrow A' \rightarrow \Box. \text{IsUmap}(R) \times \text{IsUmap}(R^{-1}).
\end{aligned}$$

4.3 Annotated type theory

We are now ready to generalize the relational interpretation of types provided by the univalent parametricity translation, so as to allow for interpreting sorts with instances of weaker structures than equivalence. For this purpose, we introduce a variant CC_{ω}^{+} of CC_{ω} where each universe is annotated with a label indicating the structure available on its relational interpretation. Recall from Section 3.2 that we have used annotations $\alpha \in \mathcal{A}$ to identify the different structures of the lattice disassembling type equivalence: these are the labels annotating sorts of CC_{ω}^{+} , so that if A has type \Box^{α} , then the associated relation A_R has type $\boxtimes^{\alpha} A A'$. The syntax of CC_{ω}^{+} is thus:

$$\begin{aligned}
M, N, A, B \in \mathcal{T}_{CC_{\omega}^{+}} &::= \Box_i^{\alpha} \mid x \mid M N \mid \lambda x : A. M \mid \Pi x : A. B \\
\alpha \in \mathcal{A} &= \{0, 1, 2_a, 2_b, 3, 4\}^2 \quad i \in \mathbb{N}
\end{aligned}$$

Before completing the actual formal definition of the TROCQ proof transfer framework, let us informally illustrate how these annotations shall drive the interpretation of terms, and in particular, of a dependent product $\Pi x : A. B$. In this case, before translating B , three terms representing the bound variable x , its translation x' , and the parametricity witness x_R are added to the context. The type of x_R is $\text{rel}(A_R) x x'$ where A_R is the parametricity witness relating A to its translation A' . The role of annotation α on the sort typing type A is thus to govern the amount of information available in witness x_R , by determining the type of A_R . This intent is reflected in the typing rules of CC_ω^+ , which rely on the definition of the loci \mathcal{D}_\square , \mathcal{D}_\rightarrow and \mathcal{D}_Π , introduced in §3.3.

Contexts are defined as usual, but typing terms in CC_ω^+ requires defining a *subtyping* relation \preccurlyeq , defined by the rules of Figure 5. The typing rules of CC_ω^+ are available in Figure 6 and follow standard presentations [6]. The \equiv relation in the (SUBCONV) rule is the *conversion* relation, defined as the closure of α -equivalence and β -reduction. The two types of judgment in CC_ω^+ are thus:

$$\Gamma \vdash_+ A \preccurlyeq B \quad \text{and} \quad \Gamma \vdash_+ M : A$$

where M, A and B are terms in CC_ω^+ and Γ is a context in CC_ω^+ .

$$\begin{array}{c} \frac{\Gamma \vdash_+ A : K \quad \Gamma \vdash_+ B : K \quad A \equiv B}{\Gamma \vdash_+ A \preccurlyeq B} \text{(SUBCONV)} \quad \frac{\alpha \geq \beta \quad i \leq j}{\Gamma \vdash_+ \square_i^\alpha \preccurlyeq \square_j^\beta} \text{(SUBSORT)} \\[10pt] \frac{\Gamma \vdash_+ M' N : K \quad \Gamma \vdash_+ M \preccurlyeq M'}{\Gamma \vdash_+ M N \preccurlyeq M' N} \text{(SUBAPP)} \\[10pt] \frac{\Gamma, x : A \vdash_+ M \preccurlyeq M'}{\Gamma \vdash_+ \lambda x : A. M \preccurlyeq \lambda x : A. M'} \text{(SUBLAM)} \\[10pt] \frac{\Gamma \vdash_+ \Pi x : A. B : \square_i \quad \Gamma \vdash_+ A' \preccurlyeq A \quad \Gamma, x : A' \vdash_+ B \preccurlyeq B'}{\Gamma \vdash_+ \Pi x : A. B \preccurlyeq \Pi x : A'. B'} \text{(SUBPI)} \\[10pt] K ::= \square_i \mid \Pi x : A. K \end{array}$$

Fig. 5: Subtyping rules for CC_ω^+

Due to space constraints, we omit the direct proof that CC_ω^+ is a conservative extension over CC_ω . It goes by defining an erasure function for terms $|\cdot|^- : \mathcal{T}_{CC_\omega^+} \rightarrow \mathcal{T}_{CC_\omega}$ and the associated erasure function for contexts.

4.4 The TROCQ calculus

The final stage of the announced generalization consists in building an analogue to the parametricity translations available in pure type systems, but for the annotated type theory of § 4.3. This analogue is geared towards proof transfer, as discussed in § 2.1, and therefore designed to *synthesize* the output of the translation from its input, rather than to *check* that certain pairs of terms are

$$\begin{array}{c}
\frac{\Gamma \vdash_+ M : A \quad \Gamma \vdash_+ A \preceq B}{\Gamma \vdash_+ M : B} (\text{CONV}^+) \quad \frac{(\alpha, \beta) \in \mathcal{D}_\square}{\Gamma \vdash_+ \square_i^\alpha : \square_{i+1}^\beta} (\text{SORT}^+) \\
\\
\frac{(x, A) \in \Gamma \quad \Gamma \vdash_+}{\Gamma \vdash_+ x : A} (\text{VAR}^+) \quad \frac{\Gamma \vdash_+ A : \square_i \quad x \notin \text{Var}(\Gamma)}{\Gamma, x : A \vdash_+} (\text{CONTEXT}^+) \\
\\
\frac{\Gamma \vdash_+ M : \Pi x : A. B \quad \Gamma \vdash_+ N : A}{\Gamma \vdash_+ M N : B[x := N]} (\text{APP}^+) \quad \frac{\Gamma, x : A \vdash_+ M : B}{\Gamma \vdash_+ \lambda x : A. M : \Pi x : A. B} (\text{LAM}^+) \\
\\
\frac{\Gamma \vdash_+ A : \square_i^\alpha \quad \Gamma \vdash_+ B : \square_i^\beta \quad \mathcal{D}_\rightarrow(\gamma) = (\alpha, \beta)}{\Gamma \vdash_+ A \rightarrow B : \square_i^\gamma} (\text{ARROW}^+) \\
\\
\frac{\Gamma \vdash_+ A : \square_i^\alpha \quad \Gamma, x : A \vdash_+ B : \square_i^\beta \quad \mathcal{D}_\Pi(\gamma) = (\alpha, \beta)}{\Gamma \vdash_+ \Pi x : A. B : \square_i^\gamma} (\text{PI}^+)
\end{array}$$

Fig. 6: Typing rules for CC_ω^+

in relation. However, splitting up the interpretation of universes into a lattice of possible relation structures means that the source term of the translation is not enough to characterize the desired output: the translation needs to be informed with some extra information about the expected outcome of the translation. In the TROCQ calculus, this extra information is a type of CC_ω^+ .

We thus define TROCQ *contexts* as lists of quadruples:

$$\Delta ::= \varepsilon \mid \Delta, x @ A \sim x' \vdash x_R \quad \text{where } A \in \mathcal{T}_{CC_\omega^+},$$

and introduce a conversion function γ from TROCQ contexts to CC_ω^+ contexts:

$$\begin{aligned}
\gamma(\varepsilon) &= \varepsilon \\
\gamma(\Delta, x @ A \sim x' \vdash x_R) &= \gamma(\Delta), x : A
\end{aligned}$$

Now, a TROCQ judgment is a 4-ary relation of the form $\Delta \vdash_t M @ A \sim M' \vdash M_R$, which is read *in context Δ , term M of annotated type A translates to term M' , because M_R and M_R is called a parametricity witness*. TROCQ judgments are defined by the rules of Figure 7. This definition involves a weakening function for parametricity witnesses, defined as follows.

Definition 9. For all $p, q \in \{0, 1, 2_a, 2_b, 3, 4\}$, such that $p \geq q$, we define map $\downarrow_q^p: M_p \rightarrow M_q$, which forgets the fields from class M_p that are not in M_q .

For all $\alpha, \beta \in \mathcal{A}$, such that $\alpha \geq \beta$, function $\Downarrow_\beta^\alpha: \boxplus^\alpha A B \rightarrow \boxplus^\beta A B$ is defined by:

$$\Downarrow_{(p,q)}^{(m,n)} \langle R, M^\rightarrow, M^\leftarrow \rangle := \langle R, \downarrow_p^m M^\rightarrow, \downarrow_q^n M^\leftarrow \rangle.$$

The weakening function on parametricity witnesses is defined on Figure 8 by extending function \Downarrow_β^α to all relevant pairs of types of CC_ω^+ , i.e., \Downarrow_U^T is defined for $T, U \in \mathcal{T}_{CC_\omega^+}$ as soon as $T \preceq U$.

$$\begin{array}{c}
\frac{(\alpha, \beta) \in \mathcal{D}_\square}{\Delta \vdash_t \square_i^\alpha @ \square_{i+1}^\beta \sim \square_i^\alpha \because p_{\square_i}^{\alpha, \beta}} \text{ (TROCCQSort)} \\
\\
\frac{(x, A, x', x_R) \in \Delta \quad \gamma(\Delta) \vdash_+}{\Delta \vdash_t x @ A \sim x' \because x_R} \text{ (TROCCQVar)} \\
\\
\frac{\Delta \vdash_t M @ \Pi x : A. B \sim M' \because M_R \quad \Delta \vdash_t N @ A \sim N' \because N_R}{\Delta \vdash_t M N @ B[x := N] \sim M' N' \because M_R N N' N_R} \text{ (TROCCQApp)} \\
\\
\frac{\Delta \vdash_t A @ \square_i^\alpha \sim A' \because A_R \quad \Delta, x @ A \sim x' \because x_R \vdash_t M @ B \sim M' \because M_R}{\Delta \vdash_t \lambda x : A. M @ \Pi x : A. B \sim \lambda x' : A'. M' \because \lambda x x' x_R. M_R} \text{ (TROCCQLam)} \\
\\
\frac{(\alpha, \beta) = \mathcal{D}_\rightarrow(\delta) \quad \Delta \vdash_t A @ \square_i^\alpha \sim A' \because A_R \quad \Delta \vdash_t B @ \square_i^\beta \sim B' \because B_R}{\Delta \vdash_t A \rightarrow B @ \square_i^\delta \sim A' \rightarrow B' \because p_{\rightarrow}^\delta A_R B_R} \text{ (TROCCQArrow)} \\
\\
\frac{(\alpha, \beta) = \mathcal{D}_\Pi(\delta) \quad \Delta \vdash_t A @ \square_i^\alpha \sim A' \because A_R \quad \Delta, x @ A \sim x' \because x_R \vdash_t B @ \square_i^\beta \sim B' \because B_R}{\Delta \vdash_t \Pi x : A. B @ \square_i^\delta \sim \Pi x' : A'. B' \because p_\Pi^\delta A_R B_R} \text{ (TROCCQPi)} \\
\\
\frac{\Delta \vdash_t M @ A \sim M' \because M_R \quad \gamma(\Delta) \vdash_+ A \preceq B}{\Delta \vdash_t M @ B \sim M' \because \Downarrow_B^A M_R} \text{ (TROCCQConv)}
\end{array}$$

Fig. 7: TROCCQ rules

An abstraction theorem relates TROCCQ judgments and typing in CC_ω^+ .

Theorem 6 (TROCCQ abstraction theorem).

$$\frac{\gamma(\Delta) \vdash_+ \quad \gamma(\Delta) \vdash_+ M : A \quad \Delta \vdash_t M @ A \sim M' \because M_R \quad \Delta \vdash_t A @ \square_i^\alpha \sim A' \because A_R}{\gamma(\Delta) \vdash_+ M' : A' \quad \text{and} \quad \gamma(\Delta) \vdash_+ M_R : \text{rel}(A_R) M M'}$$

Proof. By induction on derivation $\Delta \vdash_t M @ A \sim M' \because M_R$.

Note that type A in the typing hypothesis $\gamma(\Delta) \vdash_+ M : A$ of the abstraction theorem is exactly the extra information passed to the translation. The latter can thus also be seen as an inference algorithm, which infers annotations for the output of the translation from that of the input.

Remark 4. Since by definition of $p_{\square}^{\alpha, \beta}$ (Equation 12), we have $\vdash_t \square^\alpha @ \square^\beta \sim \square^\alpha \because p_{\square}^{\alpha, \beta}$, by applying Theorem 6 with $\gamma(\Delta) \vdash_+ A : \square^\alpha$, we get:

$$\frac{\gamma(\Delta) \vdash_+ A : \square^\alpha \quad \Delta \vdash_t A @ \square^\alpha \sim A' \because A_R}{\gamma(\Delta) \vdash_+ A_R : \text{rel}(p_{\square}^{\alpha, \beta}) A A'}.$$

$$\begin{aligned}
\Downarrow_{\square_{\alpha'}^i}^{\square_{\alpha}^i} t_R &:= \Downarrow_{\alpha'}^{\alpha} t_R & \Downarrow_{A' M'}^A M N_R &:= \Downarrow_{A'}^A M M' N_R \\
\Downarrow_{\lambda x:A'. B'}^{\lambda x:A. B} M M' N_R &:= \Downarrow_{B'[x:=M']}^{B[x:=M]} N_R \\
\Downarrow_{\Pi x:A'. B'}^{\Pi x:A. B} M_R &:= \lambda x x' x_R. \Downarrow_{B'}^B (M_R x x' (\Downarrow_A^{A'} x_R)) & \Downarrow_{A'}^A M_R &:= M_R
\end{aligned}$$

Fig. 8: Weakening of parametricity witnesses

Now by the same definition, for any $\beta \in \mathcal{A}$, $\text{rel}(p_{\square}^{\alpha, \beta}) = \boxplus^{\alpha}$, hence $\gamma(\Delta) \vdash A_R : \boxplus^{\alpha} A A'$, as expected by the type annotation $A : \square^{\alpha}$ in the premise of the rule.

Remark 5. By applying the Remark 4 with $\vdash_+ \square^{\alpha} : \square^{\beta}$, we indeed obtain that $\vdash_+ p_{\square}^{\alpha, \beta} : \boxplus^{\beta} \square^{\alpha} \square^{\alpha}$ as expected, provided that $(\alpha, \beta) \in \mathcal{D}_{\square}$.

4.5 Constants

Concrete applications require extending TROCQ with constants. Constants are similar to variables, except that they are stored in a global context instead of a typing context. A crucial difference though is that a constant may be assigned several different annotated types in CC_{ω}^+ .

Consider for example, a constant **list**, standing for the type of polymorphic lists. As **list** A is the type of lists with elements of type A , it can be annotated with type $\square^{\alpha} \rightarrow \square^{\alpha}$ for any $\alpha \in \mathcal{A}$.

Every constant declared in the global environment has an associated collection of possible annotated types $T_c \subset \mathcal{T}_{CC_{\omega}^+}$. We require that all the annotated types of a same constant share the same erasure in CC_{ω} , *i.e.*, $\forall c, \forall A, \forall B, A, B \in T_c \Rightarrow |A| = |B|$. For example, $T_{\text{list}} = \{\square^{\alpha} \rightarrow \square^{\alpha} \mid \alpha \in \mathcal{A}\}$.

In addition, we provide translations $\mathcal{D}_c(A)$ for each possible annotated type A of each constant c in the global context. For example, $\mathcal{D}_{\text{list}}(\square^{(1,0)} \rightarrow \square^{(1,0)})$ is equal to $(\text{list}, \lambda A A' A_R. (\text{List.All2 } A_R, \text{List.map } (\text{map } A_R)))$, where relation **List.All2** A_R relates lists of the same length, whose elements are pair-wise related via A_R , **List.map** is the usual map function on lists and **map** $A_R : A \rightarrow A'$ extracts the *map* projection of the record A_R of type $\boxplus^{(1,0)} A A' \equiv \Sigma R. A \rightarrow A'$. Part of these translations can be generated automatically by weakening.

We describe in Figure 9 the additional rules for constants in CC_{ω}^+ and TROCQ. Note that for an input term featuring constants, an unfortunate choice of annotation may lead to a stuck translation.

$$\frac{c \in \mathcal{C} \quad A \in T_c}{\Gamma \vdash c : A} (\text{CONST}^+) \qquad \frac{\mathcal{D}_c(A) = (c', c_R)}{\Delta \vdash c @ A \sim c' \cdot c_R} (\text{TROCQCONST})$$

Fig. 9: Additional constant rules for CC_{ω}^+ and TROCQ

We describe in Figure 9 the additional rules for constants in CC_{ω}^+ and TROCQ. Note that for an input term featuring constants, an unfortunate choice of annotation may lead to a stuck translation.

5 Implementation and applications

The TROCQ plugin [13] turns the material presented in Section 4 into an actual tactic, called `trocq`, for automating proof transfer in Coq. This tactic synthesizes a new goal from the current one, as well as a proof that the former implies the latter. User-defined relations between constants, registered via specific vernacular commands, inform this synthesis. The core of the plugin implements each rule of the TROCQ calculus in the Elpi meta-programming language [17,31], on top of Coq libraries formalizing the contents of Section 3. In the logic programming paradigm of Elpi, each rule of Figure 7 translates gracefully into a corresponding λ Prolog predicate, making the corresponding source code very close to the presentation of §4.4. However, the TROCQ plugin also implements a much less straightforward annotation inference algorithm, so as to hide the management of sort annotations to Coq users completely. This section illustrates the usage of the `trocq` tactic on various concrete examples.

5.1 Isomorphisms

Bitvectors (code). Here are two possible encodings of bitvectors in Coq:

```
bounded_nat (k : nat) := {n : nat & n < pow 2 k}. (* n < 2^k *)
bitvector (k : nat) := Vector.t Bool k. (* size k vectors of booleans *)
```

We can prove that these representations are equivalent by combining two proofs by transitivity: the proof that `bounded_nat k` is related to `bitvector k` for a given `k`, and the proof that `Vector.t` is related to itself. We also make use of the equivalence relations `natR` and `boolR`, which respectively relate type `nat` and `Bool` with themselves:

```
Rk : ∀ (k : nat), Param44.Rel (bounded_nat k) (bitvector k)
vecR : ∀ (A A' : Type) (AR : Param44.Rel A A') (k k' : nat)
  (kR : natR k k'), Param44.Rel (Vector.t A k) (Vector.t A' k')
(* equivalence between types (bounded_nat k) and (bitvector k') *)
bvR : ∀ (k k' : nat) (kR : natR k k'),
  Param44.Rel (bounded_nat k) (bitvector k')
(* informing TrocQ with these equivalences *)
TrocQ Use vecR natR boolR bvR.
```

Now, suppose we would like to transfer the following result from the bounded natural numbers to the vector-based encoding:

```
∀ (k : nat) (v : bounded_nat k) (i : nat) (b : Bool), i < k ->
  get (set v i b) i = b
```

As this goal involves `get` and `set` operations on bitvectors, and the order and equality relations on type `nat`, we inform TROCQ with the associated operations `getv` and `setv` on the vector encoding. E.g., for `get` and `getv`, we prove:

```

getR : ∀ (k k' : nat) (kR : natR k k')
  (v : bounded_nat k) (v' : bitvector k') (vR : bvR k k' kR v v')
  (n n' : nat) (nR : natR n n'), boolR (get v n) (getv v' n')

```

We can now use proof transfer from bitvectors to bounded natural numbers:

```

Trocq Use eqR ltR. (* where eq and lt are translated to themselves *)
Trocq Use getR setR.

Lemma setBitGetSame : ∀ (k : nat) (v : bitvector k),
  ∀ (i : nat) (b : Bool), i < k → getv (setv v i b) i = b.
Proof. trocq. exact setBitGetSame'. (* same lemma, on bitvector *) Qed.

```

Induction principle on integers. (code). Recall that the problem from Example 1 is to obtain the following elimination scheme, from that available on type \mathbb{N} :

```

N_ind : ∀ P : N → □, P 0_N → (∀ n : N, P n → P (S_N n)) → ∀ n : N, P n

```

We first inform TROCQ that \mathbb{N} and \mathbb{N} are isomorphic, by providing proofs that the two conversions $\uparrow_{\mathbb{N}} : \mathbb{N} \rightarrow \mathbb{N}$ and $\downarrow_{\mathbb{N}} : \mathbb{N} \rightarrow \mathbb{N}$ are mutual inverses. Using lemma `Iso.toParam`, we can deduce an equivalence `Param44.Rel N N`, i.e., $\square^{(4,4)} \mathbb{N} \mathbb{N}$. We also prove and register that zeros and successors are related:

```

Definition N_R : Param44.Rel N N := ...
Lemma 0_R : rel N_R 0_N 0_N.
Lemma S_R : ∀ m n, rel N_R m n → rel N_R (S_N m) (S_N n).
Trocq Use N_R 0_R S_R.

```

TROCQ is now able to generate, prove and apply the desired implication:

```

Lemma N_ind : ∀ P : N → □, P 0_N → (∀ n : N, P n → P (S_N n)) →
  ∀ n : N, P n.
Proof.
  trocq. (* in the goal, N, 0_N, S_N have been replaced by N, 0_N, S_N *)
  exact nat_rect.
Qed.

```

Inspecting this proof confirms that only information up to level $(2_a, 3)$ has been extracted from the equivalence proof `N_R`. It is thus possible to run the exact proof transfer, but with a weaker relation, as illustrated in the code for an abstract type I with a zero and a successor constants, and a retraction $\mathbb{N} \rightarrow I$.

5.2 Sections, retractions

Modular arithmetic (code). A typical application of modular arithmetic is to show that some statement on \mathbb{Z} can be reduced to statements on $\mathbb{Z}/p\mathbb{Z}$. Let us show how TROCQ can synthesize and prove the following implication:

```

Lemma IntRedModZp : (forall (m n p : Zmodp), (m = n * n)%Zmodp -> m = 0)
-> forall (m n p : int), (m = n * n)%int -> (m == 0)%int.
Proof. intro Hyp. trocq; simpl. now apply Hyp. Qed.

```

where scope `%Zmodp` is for the usual product and zero on type `Zmodp`, for $\mathbb{Z}/p\mathbb{Z}$, scope `%int` for those on type `int`, for \mathbb{Z} , and `==` is an equality test modulo p on type `int`. Observe that the implication deduces a lemma on \mathbb{Z} from its modular analogues. Type `Zmodp` and `int` are obviously not equivalent, but a *retraction* is actually enough for this proof transfer. We have:

```

modp : int -> Zmodp
reprp : Zmodp -> int
reprpK : ∀ (x : Zmodp), modp (reprp x) = x
Rp : Param42a.Rel int Zmodp

```

where `Rp`, (a proof that $\boxtimes^{(4,2a)} \mathbb{Z} \mathbb{Z}/p\mathbb{Z}$), is obtained from `reprpK` via lemma `SplitSurj.toParam`. Proving lemma `IntRedModZp` by `trocq` now just requires relating the respective zeroes, multiplications, and equalities of the two types:

```

R0 : Rp 0%int 0%Zmodp.
Rmul : ∀ (m : int) (x : Zmodp) (xR : Rp m x)
  (n : int) (y : Zmodp) (yR : Rp n y), Rp (m * n)%int (x * y)%Zmodp.
Reqmodp : ∀ (m : int) (x : Zmodp), Rp m x ->
  ∀ (n : int) (y : Zmodp), Rp n y -> Param01.Rel (m == n) (x = y).
Trocq Use Rp Rmul R0 Reqmodp. (* informing Trocq with these relations *)

```

where `Param01.Rel P Q` (`Param01.Rel` is the Coq name for $\boxtimes^{(0,1)}$) is $Q \rightarrow P$. Note that by definition of the relation given by `Rp`, lemma `Rmul` amounts to:

$$\forall (m n : \text{int}), \text{modp } (m * n)\% \text{int} = (\text{modp } m * \text{modp } n)\% \text{Zmodp}.$$

Summable sequences. (code). Example 2 involves two instances of subtypes: type $\overline{\mathbb{R}}_{\geq 0}$ extends a type $\mathbb{R}_{\geq 0}$ of positive real numbers with an abstract element and type `summable` is for provably convergent sequences of positive real numbers:

```

Inductive  $\overline{\mathbb{R}}_{\geq 0}$  : Type := Fin :  $\mathbb{R}_{\geq 0} \rightarrow \overline{\mathbb{R}}_{\geq 0}$  | Inf :  $\overline{\mathbb{R}}_{\geq 0}$ .
Definition seq $_{\mathbb{R}_{\geq 0}}$  := nat  $\rightarrow \mathbb{R}_{\geq 0}$ . Definition seq $_{\overline{\mathbb{R}}_{\geq 0}}$  := nat  $\rightarrow \overline{\mathbb{R}}_{\geq 0}$ .
Record summable := {to_seq :> seq $_{\mathbb{R}_{\geq 0}}$ ; _ : isSummable to_seq}.

```

Type $\overline{\mathbb{R}}_{\geq 0}$ and $\mathbb{R}_{\geq 0}$ are related at level $(4, 2_b)$: e.g., `truncate` : $\overline{\mathbb{R}}_{\geq 0} \rightarrow \mathbb{R}_{\geq 0}$ provides a partial inverse to the `Fin` injection by sending the extra `Inf` to zero. Types `summable` and `seq $_{\overline{\mathbb{R}}_{\geq 0}}$` are also related at level $(4, 2_b)$, via the relation:

```

Definition Rrseq (u : summable) (v : seq $_{\overline{\mathbb{R}}_{\geq 0}}$ ) : Type := seq_extend u = v.

```

where `seq_extend` transforms a summable sequence into a sequence of extended positive reals in the obvious way. Now $\Sigma_{\overline{\mathbb{R}}_{\geq 0}} u : \overline{\mathbb{R}}_{\geq 0}$ is the sum of a sequence

$u : \text{seq}_{\mathbb{R}_{\geq 0}}$ of extended positive reals, and we also define the sum of a sequence of positive reals, as a positive real, again by defaulting infinite sums to zero. For the purpose of the example, we only do so for summable sequences:

Definition $\Sigma_{\mathbb{R}_{\geq 0}} (u : \text{summable}) : \mathbb{R}_{\geq 0} := \text{truncate } (\Sigma_{\mathbb{R}_{\geq 0}} (\text{seq_extend } u))$.

These two notions of sums are related via `Rrseq`, and so are the respective additions of positive (resp. extended positive) reals and the respective pointwise additions of sequences. Once TROCQ is informed of these relations, the tactic is able to transfer the statement from the much easier variant on extended reals:

```
(* relating type  $\mathbb{R}_{\geq 0}$  and  $\overline{\mathbb{R}}_{\geq 0}$  and their respective equalities *)
Trocq Use Param01_paths Param42b_nnR.
(* relating sequence types, sums, addition, addition of sequences *)
Trocq Use Param4a_rseq R_sum_xnnR R_add_xnnR seq_nnR_add.

Lemma sum_xnnR_add :  $\forall (u \ v : \overline{\mathbb{R}}_{\geq 0}), \Sigma_{\overline{\mathbb{R}}_{\geq 0}} (u + v) = \Sigma_{\overline{\mathbb{R}}_{\geq 0}} u + \Sigma_{\overline{\mathbb{R}}_{\geq 0}} v$ .
Proof. (...) Qed. (* easy, as no convergence proof is needed *)

Lemma sum_nnR_add :  $\forall (u \ v : \mathbb{R}_{\geq 0}), \Sigma_{\mathbb{R}_{\geq 0}} (u + v) = \Sigma_{\mathbb{R}_{\geq 0}} u + \Sigma_{\mathbb{R}_{\geq 0}} v$ .
Proof. trocq; exact sum_xnnR_add. Qed.
```

5.3 Polymorphic, dependent types

Polymorphic parameters (code). Suppose we want to transfer a goal involving lists along an equivalence between the types of the values contained in the lists. We first prove that the `list` type former is equivalent to itself, and register this fact:

```
listR :  $\forall A \ A' (AR : \text{Param44.Rel } A \ A'), \text{Param44.Rel } (\text{list } A) (\text{list } A')$ 
Trocq Use listR.
```

We also need to relate with themselves all operations on type `list` involved in the goal, including constructors, and to register these facts, before TROCQ is able to transfer any goal, *e.g.*, about `list N` to its analogue on `list N`.

Note that lemma `listR` requires an *equivalence* between its parameters. If this does not hold, as in the case of type `int` and `Zmodp` from Section 5.1, the translation is stuck: weakening does not apply here. In order to avoid stuck translation, we need several versions of `listR` to cover all cases. For instance, the following lemma is required for proof transfers from `list Zmodp` to `list int`.

```
listR2a4 :  $\forall A \ A' (AR : \text{Param2a4.Rel } A \ A'),$ 
   $\text{Param2a4.Rel } (\text{list } A) (\text{list } A')$ .
```

Dependent and polymorphic types (code). Fixed-size vectors can be represented by iterated tuples, an alternative to the inductive type `Vector.t`, from Coq's standard library, as follows.

```

Definition tuple (A : Type) : nat -> Type := fix F n :=
  match n with 0 => Unit | S n' => F n' * A end.

```

On the following mockup example, TROCQ transfers a lemma on `Vector.t` to its analogue on `tuple`, about a function `head : $\forall A n, \text{tuple } A (S n) \rightarrow A$` , and a function `const : $\forall A, A \rightarrow \forall n, \text{tuple } A n$` creating a constant vector, and simultaneously refines integers into the integers modulo p from Section 5.1:

```

Lemma head_cst (n : nat) (i : int) : Vector.hd (Vector.const i (S n)) = i.
Proof. destruct n; simpl; reflexivity. Qed. (* easy proof *)

Lemma head_cst' :  $\forall (n : nat) (z : \mathbb{Z}_{\text{mod } p}), \text{head } (\text{const } z (S n)) = z$ .
Proof. trocq. exact head_const. Qed.

```

This automated proof only requires proving (and registering) that `head` and `const` are related to their analogue `Vector.hd` and `Vector.const`, from Coq's standard library. Note that the proof uses the equivalence between `Vector.t` and `tuple` but only requires a retraction between parameter types.

6 Conclusion

The TROCQ framework can be seen as a generalization of the univalent parametricity translation [30]. It allows for weaker relations than equivalence, thanks to a fine-grained control of the data propagated by the translation. This analysis is enabled by skolemizing the usual symmetrical presentation of equivalence, so as to expose the data, and by introducing a hierarchy of algebraic structures for relations. This scrutiny allows in particular to get rid of the univalence axiom for a larger class of equivalence proofs [29], and to deal with refinement relations for arbitrary terms, unlike the CoqEAL library [14]. Altenkirch and Kaposi already proposed a symmetrical, skolemized phrasing of type equivalence [3], but for different purposes. In particular, they did not study the resulting hierarchy of structures. Definition 4 however slightly differs from theirs: by reducing the amount of transport involved, it eases formal proofs significantly in practice, both in the internal library of TROCQ and for end-users of the tactic.

The concrete output of this work is a plugin [13] that consists of about 3000 l. of original Coq proofs and 1200 l. of meta-programming, in the Elpi meta-language, excluding white lines and comments. This plugin goes beyond the state of the art in two ways. First, it demonstrates that a single implementation of this parametricity framework covers the core features of several existing other tactics, for refinements [14,16], generalized rewriting [28], and proof transfer [30]. Second, it addresses use cases, such as Example 2, that are beyond the skills of any existing tool in any proof assistant based on type theory. The prototype plugin arguably needs an improved user interface so as to reach the maturity of some of the aforementioned existing tactics. It would also benefit from an automated generation of equivalence proofs, such as Pumpkin Pi [27].

Acknowledgments. The authors would like to thank András Kovács, Kenji Maillard, Enrico Tassi, Quentin Vermande, Théo Winterhalter, and anonymous reviewers, whose comments greatly helped us improve this article.

Disclosure of Interests. None.

References

1. The lean mathematical library. In: Blanchette, J., Hritcu, C. (eds.) *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*. pp. 367–381. ACM (2020). <https://doi.org/10.1145/3372885.3373824>, <https://doi.org/10.1145/3372885.3373824>
2. Affeldt, R., Cohen, C.: Measure construction by extension in dependent type theory with application to integration (2023), accepted for publication in JAR
3. Altenkirch, T., Kaposi, A.: Towards a cubical type theory without an interval. In: *TYPES. LIPIcs*, vol. 69, pp. 3:1–3:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2015)
4. Angiuli, C., Brunerie, G., Coquand, T., Harper, R., (Favonia), K.H., Licata, D.R.: Syntax and models of cartesian cubical type theory. *Math. Struct. Comput. Sci.* **31**(4), 424–468 (2021). <https://doi.org/10.1017/S0960129521000347>, <https://doi.org/10.1017/S0960129521000347>
5. Angiuli, C., Cavallo, E., Mörtberg, A., Zeuner, M.: Internalizing representation independence with univalence. *Proc. ACM Program. Lang.* **5**(POPL), 1–30 (2021). <https://doi.org/10.1145/3434293>, <https://doi.org/10.1145/3434293>
6. Aspinall, D., Compagnoni, A.B.: Subtyping dependent types. *Theor. Comput. Sci.* **266**(1-2), 273–309 (2001). [https://doi.org/10.1016/S0304-3975\(00\)00175-4](https://doi.org/10.1016/S0304-3975(00)00175-4), [https://doi.org/10.1016/S0304-3975\(00\)00175-4](https://doi.org/10.1016/S0304-3975(00)00175-4)
7. Barthe, G., Capretta, V., Pons, O.: Setoids in type theory. *J. Funct. Program.* **13**(2), 261–293 (2003). <https://doi.org/10.1017/S0956796802004501>, <https://doi.org/10.1017/S0956796802004501>
8. Bauer, A., Gross, J., Lumsdaine, P.L., Shulman, M., Sozeau, M., Spitters, B.: The hott library: a formalization of homotopy type theory in coq. In: Bertot, Y., Vafeiadis, V. (eds.) *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*. pp. 164–172. ACM (2017). <https://doi.org/10.1145/3018610.3018615>, <https://doi.org/10.1145/3018610.3018615>
9. Bernardy, J., Jansson, P., Paterson, R.: Proofs for free - parametricity for dependent types. *J. Funct. Program.* **22**(2), 107–152 (2012). <https://doi.org/10.1017/S0956796812000056>, <https://doi.org/10.1017/S0956796812000056>
10. Bernardy, J., Lasson, M.: Realizability and parametricity in pure type systems. In: Hofmann, M. (ed.) *Foundations of Software Science and Computational Structures - 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings. Lecture Notes in Computer Science*, vol. 6604, pp. 108–122. Springer (2011). https://doi.org/10.1007/978-3-642-19805-2_8, https://doi.org/10.1007/978-3-642-19805-2_8
11. Boulrier, S., Pédrot, P., Tabareau, N.: The next 700 syntactical models of type theory. In: Bertot, Y., Vafeiadis, V. (eds.) *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*. pp. 182–194. ACM (2017). <https://doi.org/10.1145/3018610.3018620>, <https://doi.org/10.1145/3018610.3018620>
12. Cohen, C., Coquand, T., Huber, S., Mörtberg, A.: Cubical type theory: A constructive interpretation of the univalence axiom. *FLAP* **4**(10), 3127–3170 (2017), <http://collegepublications.co.uk/ifcolog/700019>

13. Cohen, C., Crance, E., Mahboubi, A.: coq-community/trocq: Trocq 0.1.5 (Jan 2024). <https://doi.org/10.5281/zenodo.10563382>, <https://doi.org/10.5281/zenodo.10563382>
14. Cohen, C., Dénès, M., Mörtberg, A.: Refinements for free! In: Gonthier, G., Norrish, M. (eds.) *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*. Lecture Notes in Computer Science, vol. 8307, pp. 147–162. Springer (2013). https://doi.org/10.1007/978-3-319-03545-1_10, https://doi.org/10.1007/978-3-319-03545-1_10
15. Coquand, T., Huet, G.P.: The calculus of constructions. *Inf. Comput.* **76**(2/3), 95–120 (1988). [https://doi.org/10.1016/0890-5401\(88\)90005-3](https://doi.org/10.1016/0890-5401(88)90005-3), [https://doi.org/10.1016/0890-5401\(88\)90005-3](https://doi.org/10.1016/0890-5401(88)90005-3)
16. Dénès, M., Mörtberg, A., Siles, V.: A refinement-based approach to computational algebra in coq. In: Beringer, L., Felty, A. (eds.) *Interactive Theorem Proving*. pp. 83–98. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
17. Dunchev, C., Guidi, F., Coen, C.S., Tassi, E.: ELPI: fast, embeddable, λ prolog interpreter. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*. Lecture Notes in Computer Science, vol. 9450, pp. 460–468. Springer (2015). https://doi.org/10.1007/978-3-662-48899-7_32, https://doi.org/10.1007/978-3-662-48899-7_32
18. Gouëzel, S.: Vitali-carathéodory theorem in mathlib. https://leanprover-community.github.io/mathlib_docs/measure_theory/integral/vitali_caratheodory.html (2021)
19. Keller, C., Lasson, M.: Parametricity in an impredicative sort. In: Cégielski, P., Durand, A. (eds.) *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France. LIPIcs*, vol. 16, pp. 381–395. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2012). <https://doi.org/10.4230/LIPIcs.CSL.2012.381>, <https://doi.org/10.4230/LIPIcs.CSL.2012.381>
20. Martin-Dorel, É., Melquiond, G.: Proving tight bounds on univariate expressions with elementary functions in coq. *J. Autom. Reason.* **57**(3), 187–217 (2016). <https://doi.org/10.1007/s10817-015-9350-4>, <https://doi.org/10.1007/s10817-015-9350-4>
21. Mitchell, J.C.: Representation independence and data abstraction. In: *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, USA, January 1986. pp. 263–276. ACM Press (1986). <https://doi.org/10.1145/512644.512669>, <https://doi.org/10.1145/512644.512669>
22. de Moura, L., Ullrich, S.: The lean 4 theorem prover and programming language. In: Platzer, A., Sutcliffe, G. (eds.) *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*. Lecture Notes in Computer Science, vol. 12699, pp. 625–635. Springer (2021). https://doi.org/10.1007/978-3-030-79876-5_37, https://doi.org/10.1007/978-3-030-79876-5_37
23. Nederpelt, R., Geuvers, H.: *Type Theory and Formal Proof: An Introduction*. Cambridge University Press (2014). <https://doi.org/10.1017/CB09781139567725>
24. Norell, U.: Dependently typed programming in agda. In: Koopman, P.W.M., Plasmeijer, R., Swierstra, S.D. (eds.) *Advanced Functional Programming*, 6th

- International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures. Lecture Notes in Computer Science, vol. 5832, pp. 230–266. Springer (2008). https://doi.org/10.1007/978-3-642-04652-0_5, https://doi.org/10.1007/978-3-642-04652-0_5
25. Paulin-Mohring, C.: Introduction to the Calculus of Inductive Constructions. In: Paleo, B.W., Delahaye, D. (eds.) *All about Proofs, Proofs for All, Studies in Logic (Mathematical logic and foundations)*, vol. 55. College Publications (Jan 2015), <https://inria.hal.science/hal-01094195>
 26. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: Mason, R.E.A. (ed.) *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*. pp. 513–523. North-Holland/IFIP (1983)
 27. Ringer, T., Porter, R., Yazdani, N., Leo, J., Grossman, D.: Proof repair across type equivalences. In: Freund, S.N., Yahav, E. (eds.) *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. pp. 112–127. ACM (2021). <https://doi.org/10.1145/3453483.3454033>, <https://doi.org/10.1145/3453483.3454033>
 28. Sozeau, M.: A new look at generalized rewriting in type theory. *J. Formaliz. Reason.* **2**(1), 41–62 (2009). <https://doi.org/10.6092/issn.1972-5787/1574>, <https://doi.org/10.6092/issn.1972-5787/1574>
 29. Tabareau, N., Tanter, É., Sozeau, M.: Equivalences for free: univalent parametricity for effective transport. *Proceedings of the ACM on Programming Languages* **2**(ICFP), 1–29 (2018)
 30. Tabareau, N., Tanter, É., Sozeau, M.: The marriage of univalence and parametricity. *Journal of the ACM (JACM)* **68**(1), 1–44 (2021)
 31. Tassi, E.: Deriving proved equality tests in Coq-elpi: Stronger induction principles for containers in Coq. In: *ITP 2019 - 10th International Conference on Interactive Theorem Proving*. Portland, United States (Sep 2019). <https://doi.org/10.4230/LIPIcs.CVIT.2016.23>, <https://inria.hal.science/hal-01897468>
 32. The Coq Development Team: The coq proof assistant (Sep 2022). <https://doi.org/10.5281/zenodo.7313584>, <https://doi.org/10.5281/zenodo.7313584>
 33. Univalent Foundations Program, T.: *Homotopy Type Theory: Univalent Foundations of Mathematics*. <https://homotopytypetheory.org/book>, Institute for Advanced Study (2013)
 34. Vezzosi, A., Mörtberg, A., Abel, A.: Cubical agda: a dependently typed programming language with univalence and higher inductive types. *Proc. ACM Program. Lang.* **3**(ICFP), 87:1–87:29 (2019). <https://doi.org/10.1145/3341691>, <https://doi.org/10.1145/3341691>
 35. Wadler, P.: Theorems for free! In: Stoy, J.E. (ed.) *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*. pp. 347–359. ACM (1989). <https://doi.org/10.1145/99370.99404>, <https://doi.org/10.1145/99370.99404>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

