Trocq: Proof Transfer for Free, Beyond Equivalence and Univalence

Cyril Cohen, Enzo Crance, Assia Mahboubi

Trocq: Proof Transfer for Free, Beyond Equivalence and Univalence

CYRIL COHEN, Inria, CNRS, ENS de Lyon, Université Claude Bernard Lyon 1, LIP, UMR 5668, France
ENZO CRANCE, Nantes Université, École Centrale Nantes, CNRS, Inria, LS2N, UMR 6004, France and Mitsubishi Electric R&D Centre Europe, France
ASSIA MAHBOUBI, Nantes Université, École Centrale Nantes, CNRS, Inria, LS2N, UMR 6004, France and Vrije Universiteit Amsterdam, The Netherlands

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 Rocq/Coq interactive theorem prover, in the Coq-Elpi meta-language.

CCS Concepts: • **Theory of computation → Type theory**.

Additional Key Words and Phrases: Parametricity, Representation independence, Univalence, Proof assistants, Proof transfer

ACM Reference Format:
Cyril Cohen, Enzo Crance, and Assia Mahboubi. 2025. Trocq: Proof Transfer for Free, Beyond Equivalence and Univalence. *ACM Trans. Program. Lang. Syst.* ?, ?, Article ? (May 2025), 40 pages. https://doi.org/10.1145/3737283

1 Introduction

Machine-checked mathematics provides every object and statement of the mathematical literature with explicit data structures, in a certain choice of foundational formalism. Different application domains may involve rather different natures of data structures, and therefore sharpen different styles of proof automation: for instance, mechanized semantics of programming languages typically involves inductive types with a much larger number of constructors than formalized undergraduate mathematics, while the latter typically involves much larger hierarchies of algebraic structures than the former. However despite the tremendous progress in the technology of interactive theorem proving, writing machine-checked mathematics remains as of today a quite laborious task, plagued by the need to justify numerous proof steps that are left implicit on paper.

A quite significant amount of the bureaucracy of machine-checked proofs consists in justifying that certain changes in a formal statement do not affect its provability. Building blocks of these changes might be non-trivial, such as proving the equivalence of two types, but propagating existing relations between certain types and between certain constants, often amounts to a systematic,

Authors' Contact Information: Cyril Cohen, cyril.cohen@inria.fr, Inria, CNRS, ENS de Lyon, Université Claude Bernard Lyon 1, LIP, UMR 5668, 69342, Lyon cedex 07, France; Enzo Crance, Nantes Université, École Centrale Nantes, CNRS, Inria, LS2N, UMR 6004, F-44000 Nantes, France and Mitsubishi Electric R&D Centre Europe, Rennes, France, enzo.crance@inria.fr; Assia Mahboubi, Nantes Université, École Centrale Nantes, CNRS, Inria, LS2N, UMR 6004, F-44000 Nantes, France and Vrije Universiteit Amsterdam, Amsterdam, The Netherlands, assia.mahboubi@inria.fr.

ACM 1558-4593/2025/5-ART?
https://doi.org/10.1145/3737283

syntax-directed recombination process. Although some existing tools address specific cases of this general question, the lack of sufficiently powerful approaches for automating *proof transfer* is a trans-disciplinary formalization issue, reported by formal proof practitioners in a broad spectrum of areas, including various aspects of mechanized semantics [16, 62, 63] or discrete mathematics [3, 33].

In this paper, we focus on interactive theorem provers based on dependent type theory, such as Rocq/Coq [55, 56], Agda [46] or Lean [26]. 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 [44] and parametricity meta-theorems [48, 61]. 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 [14], inductive types [13] and scale to the Calculus of Inductive Constructions, with an impredicative sort [36].

In particular, the *univalent parametricity* translation [53] leverages the univalence axiom [57] 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 [8]. This last approach is thus particularly relevant in cubical type theory [19, 59]. Indeed, a computational interpretation of the univalence axiom brings computational adequacy to otherwise possibly stuck terms, those defined via an axiomatized univalence principle.

Yet taming the bureaucracy of proof transfer remains hard in practice for users of Rocq/Coq, Lean or Agda: different concrete tools exist for specific classes of applications, but they are incomparable in scope, may assume excessively strong axioms, and are actually unable to address simple examples of data refinements [3] or elementary proofs about monadic data structures. These limitations are in fact of a fundamental nature, and call for a more detailed study of parametricity in dependent type theory.

Contributions. This paper presents three contributions:

- A parametricity framework *à la carte*, that generalizes the univalent parametricity translation [53], as well as refinements à la CoqEAL [23] and generalized rewriting [50]. Its pivotal ingredient is a variant of Altenkirch and Kaposi's symmetrical presentation of type equivalence [5].
- A conservative subtyping extension of CC_ω [25], used to formulate an inference algorithm for this parametricity framework so as to guide the synthesis of translated terms.
- A parametricity plugin for the Rocq/Coq interactive theorem prover, which implements this parametricity framework using the Coq-Elpi [54] meta-language. This plugin rests on a library of original formal proofs, and it is distributed with a collection of application examples. It comes in two versions. The first one is built on top of the HoTT library [12] and uses the HoTT type of paths. The second one is built on top of the Rocq/Coq' standard library, and accommodates the `Prop` sort. The complete Rocq/Coq development is available at:

  https://github.com/rocq-community/trocq

Prior publications. This article is a substantial revision of a prior conference publication [21]. The paper was reshaped and presents our work from a different angle. Section 2 is new and provides a

significantly extended gallery of detailed examples. Section 4 was extended with detailed examples as well. Section 7 expands the content of the artefact report [20] by illustrating the shape of the constraint graph and presenting the recently improved user interface for the plugin. Finally, Section 8 has been updated according to the recent changes in the plugin and to the additional examples discussed in the article. The main evolution of the code is the completion of a `Prop` compatible version of the plugin.

Outline. Section 2 illustrates the state of the art of proof transfer tools in type theory. We then clarify in Sections 3 and 4 the essence of internal proof transfer in type theory, and we discuss the scope and limits of univalent parametricity. In Section 5, we present an alternative definition of type equivalence, motivating a hierarchy of structures for relations preserved by parametricity. Section 6 then presents the framework behind Trocq, as well as the synthesis of parametricity translations in this framework, before Section 7 discusses the implementation of the Trocq plugin. Section 8, revisits and extends the introductory gallery of examples. We discuss related work in Section 9 before concluding in Section 10.

2 Proof transfer by example

This section discusses a collection of concrete formalization problems in Rocq/Coq, illustrating various facets of proof transfer, as well as the related formal proof bureaucracy.

A same mathematical concept most often has several equally useful, *equivalent* formal definitions, all the more so when these formal definitions are to be mechanized. This issue is specially acute when a proof involves computational steps. In this case, a machine-checked version usually involves different data structures for the deductive parts of the proof and for the computational ones, the former being well-suited to reasoning, and the latter making computations tractable. Such implementation details are usually irrelevant on paper, but machine-checked proofs need to convince the proof checker that the representation change does not matter. Because of this tension between smooth proofs and tractable computations, the standard library of the Rocq/Coq interactive theorem prover [55, 56] features two data structures for representing non-negative integers. Type $\mathbb{N}$ is the base-1 number system and the associated elimination principle `ℕ_ind` coincides with the usual recurrence scheme of Peano arithmetic:

```
Inductive ℕ : Type := O_ℕ : ℕ | S_ℕ (n : ℕ) : ℕ.

ℕ_ind : ∀ P : ℕ → □, P O_ℕ → (∀ n : ℕ, P n → P (S n)) → ∀ n : ℕ, P n
```

On the other hand, type `N` provides a binary representation `positive` of positive integers, as sequences of bits with a 1 digit in head position, and is thus better suited for coding (more) efficient arithmetic operations. The successor function `S_N : N → N` is no longer a constructor of the type, but can be implemented as a program, via an auxiliary successor function `S_pos` for type `positive`.

```
Inductive positive : Type :=
  xI : positive → positive | xO : positive → positive | xH : positive.

Inductive N : Type := O_N : N | Npos : positive → N.

Fixpoint S_pos (p : positive) : positive := match p with
  | xH ⇒ xO xH | xO p ⇒ xI p | xI p ⇒ xO (S_pos p) end.

Definition S_N (n : N) := match n with
  | Npos p ⇒ Npos (S_pos p) | _ ⇒ Npos xH end.
```

This successor function is useful to implement conversions $\uparrow_{\mathbb{N}} : \mathsf{N} \to \mathbb{N}$ and $\downarrow_{\mathbb{N}} : \mathbb{N} \to \mathsf{N}$ between the unary and binary representations. These conversion functions are in fact inverses of each other, and types `ℕ` and `N` are therefore *equivalent*.

Example 2.1 (Transfer of ground facts by equivalence). Computing the product $100 \times 101 \times 102$ illustrates how inappropriate type `ℕ` is for representing integers, as soon as a small amount of computation is needed in a proof. By contrast, the implementation of multiplication on type `N` is efficient enough for a small computational step.[1]

```
(* Two definition of the finite sequence [100; 101; 102], respectively on types ℕ and N *)
Definition l_ℕ : list ℕ := [100; 101; 102].
Definition l_N : list N := [100 ; 101; 102]%N.

(* Testing that 100 * 101 * 102 < (100 * 101 * 102) * (100 * 101 * 102) *)
(* triggers a  "Stack overflow" error. *)
Fail Eval compute in Nat.ltb (fold_right mult 1 l_ℕ) (fold_right mult 1 (l_ℕ ++ l_ℕ)).

(* Same test on type N is instantaneous. *)
Eval compute in N.ltb (fold_right N.mul 1%N l_N) (fold_right N.mul 1%N (l_N ++ l_N)).
```

Suppose that a proof requires testing that $\prod_{i \in l} i < \prod_{i \in l +\!\!+ l} i$, for l denoting the finite sequence $[100; 101; 102]$ and $+\!\!+$, the concatenation of sequences. This ground statement, phrased as a property on type `ℕ`, can be transferred to type `N` using conversions $\uparrow_{\mathbb{N}}$ and $\downarrow_{\mathbb{N}}$, as well as the fact that multiplications and comparisons are related via:

$$\forall x, y : \mathbb{N}, \quad \downarrow_{\mathbb{N}} (x \times_{\mathbb{N}} y) = (\downarrow_{\mathbb{N}} x) \times_{\mathsf{N}} (\downarrow_{\mathbb{N}} y) \qquad \forall x, y : \mathbb{N}, \quad x <_{\mathbb{N}} y \iff \downarrow_{\mathbb{N}} x <_{\mathsf{N}} \downarrow_{\mathbb{N}} y$$

Then, the result of the computation can be transferred back to type `ℕ`, again using the equivalence between the two types and their related operations.

Example 2.2 (Transfer of arbitrary properties by equivalence). On the other hand, the structural elimination scheme for type `N` generated by the proof assistant is actually seldom useful for reasoning about integers:

```
N_ind_struct : ∀ P : N → Prop, P 0_N → (∀ p : positive, P (Npos p)) → ∀ n : N, P n
```

but the usual recurrence scheme on integers can be obtained by *transferring* `ℕ_ind` to type `N`:

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

Incidentally, `N_ind` can be proved from `ℕ_ind` by using only the fact that $\downarrow_{\mathbb{N}}$ is a left inverse of $\uparrow_{\mathbb{N}}$, and the following relations between zeros and successors:

$$\downarrow_{\mathbb{N}} 0_{\mathbb{N}} = 0_{\mathsf{N}} \quad \text{and} \quad \forall n : \mathbb{N}, \quad \downarrow_{\mathbb{N}} (S_{\mathbb{N}}\ n) = S_{\mathsf{N}}\ (\downarrow_{\mathbb{N}}\ n)$$

Example 2.2 is in fact one of the motivating problems for *univalent parametricity* [52]. Assuming the univalence axiom, the associated prototype plugin is able to synthesize and prove the recurrence principle `N_ind` from the statement of `ℕ_ind`. This plugin relies on type class inference for combining registered proofs of the relations between the two types, their respective zeros and their respective successors. The older CoqEAL library [23] was specifically designed for automating refinements proofs, between proof-oriented data structures and computation oriented ones. Also based on parametricity techniques [36], it can actually synthesize the proof by round trip between type `ℕ` and type `N` of Example 2.1.

[1]This code for computing $(100 * 101 * 102)^2$ is arguably convoluted: this toy example is actually crafted so as to avoid being artificially saved by a lazy evaluation strategy or by parsing optimizations.

CoqEAL actually targets a larger class of *data refinements*, which may involve types that are not equivalent. Automating machine-checked proofs of data refinements is actually an active topic of research. Examples from the literature include lists and AVL trees [27], pair of integers and rational numbers as pairs of coprime integers [23], rational numbers as pairs of coprime integers and arbitrary precision arithmetic (cf Ocaml's `bignum`) [3], etc. For the sake of conciseness, we provide below a toy, but emblematic example of refinement.

Example 2.3 (Transfer of ground facts, beyond equivalences). Type $\mathbb{Z}_9$ is a dependent pair whose constructor `mk_`$\mathbb{Z}_9$ packs an integer n with a proof that n is smaller than 9:

```
Inductive ℤ₉ := mk_ℤ₉ : ∀ n : ℕ, n < 9 → ℤ₉.
```

We denote $\downarrow_{(9)} : \mathbb{Z}_9 \to \mathbb{N}$ the first projection of the pair, which just throws away the proof component, so that for instance $\downarrow_{(9)} 0_{\mathbb{Z}_9} = 0_{\mathbb{N}}$. We implement a reverse conversion $\uparrow_{(9)} : \mathbb{N} \to \mathbb{Z}_9$, which mods its argument by 9 and is thus a left inverse of $\downarrow_{(9)}$:

$$\forall x : \mathbb{Z}_9, \quad \uparrow_{(9)} \downarrow_{(9)} x = x$$

For the sake of conciseness and when there is no ambiguity we also write $\overline{x} = \uparrow_{(9)} x$.

Note that we assume here that the proof component is irrelevant, which can be achieved either by using a boolean comparison test in the definition of type $\mathbb{Z}_9$, or by using a proof-irrelevant sort, such as `hProp` or the recently introduced proof-irrelevant sort `SProp` [31]. Type $\mathbb{Z}_9$ can be used as the carrier type of arithmetic modulo 9, and in this case, multiplication on type $\mathbb{Z}_9$ is implemented by mod-ing the result of its analogue on type $\mathbb{N}$, so that for any $x\ y : \mathbb{N}$:

$$9 \mid x \times_{\mathbb{N}} y \quad \Leftrightarrow \quad \overline{x \times_{\mathbb{N}} y} = \overline{0} \quad \Leftrightarrow \quad \overline{x} \times_{\mathbb{Z}_9} \overline{y} = \overline{0}$$

Proving that 9 divides $23649 \times_{\mathbb{N}} 23703$ should easily reduce to the computational test that $\overline{23649} \times_{\mathbb{Z}_9} \overline{23703}$ is $\overline{0}$.

The instance of mechanism involved in Example 2.3, translating type $\mathbb{N}$ into type $\mathbb{Z}_9$ before computing, also falls in the scope of the CoqEAL library. It is indeed quite similar to that of Example 2.1, except for the observation that transferring this ground, quantifier-free statement only involves $\uparrow_{(9)}$. The CoqEAL method however suffers from annoying intrinsic limitations. In particular, it cannot deal with general dependent types, and is therefore useless for Examples 2.2 and 2.4, which involve universal quantifications.

Example 2.4 (Transfer of arbitrary properties, beyond equivalences). In elementary number theory, certain arithmetical properties on integers become easy to prove when they are reduced to their analogue on $\mathbb{Z}_n$, for a well-chosen n. The proof of Proposition 2.5, a first step to prove Fermat's last theorem for degree 3, is an emblematic example of this technique.

Proposition 2.5. *For any nonnegative integers x, y, z, if xyz is not divisible by 3 then* $x^3 + y^3 \neq z^3$.

Proof. We need to show that $\forall x, y, z \in \mathbb{N},\ 3 \nmid xyz \implies x^3 +_{\mathbb{N}} y^3 \neq z^3$. By reduction modulo 9, it suffices to show $\forall x, y, z \in \mathbb{Z}_9,\ 3 \nmid xyz \implies x^3 +_{\mathbb{Z}_9} y^3 \neq z^3$. We conclude by computing the truth value of the statement for each value of $(x, y, z) \in \mathbb{Z}_9^3$. □

This proof thus essentially proceeds in two steps: a transfer step to change $\mathbb{N}$ with $\mathbb{Z}_9$, and a computational one to compute truth values on a small finite domain. Reusing the notations from Example 2.3, the transfer step essentially consists in propagating conversions $\downarrow_{(9)}$ and $\uparrow_{(9)}$, and in using the relations between the respective addition, multiplication and divisibility by 3, on types $\mathbb{Z}_9$ and $\mathbb{N}$.

The current state of affairs is actually quite unsatisfactory. In fact, we are not aware of any tool able to automate the reduction modulo 9 of Example 2.4, in any interactive prover based on dependent type theory, and this despite the similar nature of the bureaucracy involved this example and in the previous ones in this section. Also, none of the implemented frameworks mentioned so far can treat simultaneously all of Example 2.1, Example 2.2 and Example 2.3. Indeed, CoqEAL is useless on Example 2.2, because it is inherently unable to deal with its quantifiers, but univalent parametricity cannot deal with Example 2.3, because it is inherently geared for type equivalences. Finally, up to our knowledge, all the existing methods from the literature for automating Example 2.2, such as univalent parametricity or the structure identity principle [8], pull in the univalence principle in the proof, although it can be obtained by hand by very elementary means.

Unneeded dependencies on the univalence axiom is especially unsatisfactory for developers of libraries formalizing classical mathematics, and notably Lean's mathlib or Rocq/Coq's mathcomp-analysis. These libraries indeed typically assume a strong form of proof irrelevance, which is incompatible with univalence, and thus with univalent parametricity.

In short, existing techniques for transferring results from one type to another, *e.g.*, from `ℕ` to `N` or from `ℕ` to `ℤ₉`, 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. Section 3 provides a more formal analysis of this assessment.

3 Strengths and limits of univalent parametricity

We first clarify the essence of proof transfer in dependent type theory (§ 3.1) and briefly recall a few concepts related to type equivalence and univalence (§ 3.2). We then review and discuss the limits of univalent parametricity (§ 3.3).

3.1 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 $\square_i$:

$$\mathcal{T}_{CC_\omega} \ni A, B, t, u ::= \square_i \mid x \mid t\ u \mid \lambda x : A.\, M \mid \Pi x : A.\, B$$

We omit the typing rules of the calculus, and refer the reader to standard references (*e.g.*, [45, 47]). We also use the standard equality type, called propositional equality, as well as dependent pairs, denoted $\Sigma x : A.\, B$. We write $t \equiv u$ for the definitional equality between two terms t and u. Interactive theorem provers like Rocq/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 $\square$. Contexts of CC_ω, denoted $\mathcal{C}_{CC_\omega}$, are association lists of variables and types: $\mathcal{C}_{CC_\omega} \ni \Gamma ::= \langle\rangle \mid \Gamma, x : A$.

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

$$\Gamma \vdash w : \forall (s : S)(t : T), R_T\ s\ t \to R_\square (V\ s)(W\ t)$$

for a suitable context $\Gamma \in \mathcal{C}_{CC_\omega}$. 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.*, $R_\square\ A\ B \triangleq B \to A$, as in this case the proof w substantiates a proof step turning a goal clause $\Gamma \vdash V\ s$ into $\Gamma \vdash W\ t$.

Phrased as such, this synthesis problem is arguably quite loosely specified. Consider for instance the transfer problem discussed in Example 2.2. A first possible formalization involves the design of

an appropriate common interface structure for types $\mathbb{N}$ and N, for instance by setting both S and T as $\Sigma N : \square.N \times (N \to N)$, and both V and W as: $\lambda X : S.\, \Pi P : X.1 \to \square.\, P\ X.2 \to (\Pi n : X.1.\, P\ n \to P\ (X.3\ n)) \to \Pi n : X.1.\, P\ n$, 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 [8]. In the context of CC_ω, the hassle of devising explicit structures by hand has been termed the *anticipation* problem [53].

Another option is to consider two different types $S_1 \triangleq \mathbb{N} \times (\mathbb{N} \to \mathbb{N})$ and $T_1 \triangleq \mathrm{N} \times (\mathrm{N} \to \mathrm{N})$ and the following type formers V_1 and W_1:

$$V_1 \triangleq \lambda X : S_1.\ \forall P : \mathbb{N} \to \square.\ P\ X.1 \to (\forall n : \mathbb{N}, P\ n \to P(X.2\ n)) \to \forall n : \mathbb{N}, P\ n$$
$$W_1 \triangleq \lambda X : T_1.\ \forall P : \mathrm{N} \to \square.\ P\ X.1 \to (\forall n : \mathrm{N}, P\ n \to P(X.2\ n)) \to \forall n : \mathrm{N}, P\ n$$

where $X.1$ (resp $X.2$) denotes the first (resp. second) projection of the pair X. In this case, one would typically expect R_T to be a type equivalence between S_1 and T_1, so as to transport $(V_1\ s)$ to $(W_1\ t)$, 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 two types S_2 and T_2, with $R_T\ s\ t$ defined as $s = \phi\ t$, for some $\phi : T_2 \to S_2$. 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:

$$W_2 \triangleq \lambda X : T_1.\quad \forall P : \mathbb{N} \to \square.\ P\ (\uparrow_{\mathbb{N}} X.1) \to (\forall n : \mathbb{N}, P\ n \to P\ (\uparrow_{\mathbb{N}} (X.2\ (\downarrow_{\mathbb{N}} n)))) \to \forall n : \mathbb{N}, P\ n.$$

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 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* [50], which provides essential support to formalizations based on setoids [10], addresses the case of homogeneous (and reflexive) relations, *i.e.*, when S and T coincide. The CoqEAL library [23] 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.

3.2 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 [57]. Two functions $f, g : A \to B$ 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 3.1 (Type isomorphism, type equivalence). A function $f : A \to B$ is an *isomorphism*, denoted $\mathsf{IsIso}(f)$, if there exists a function $g : B \to A$ 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 $\mathsf{IsEquiv}(f)$, when it moreover enjoys a *coherence property*, relating the proofs of the section and retraction properties and ensuring that $\mathsf{IsEquiv}(f)$ is proof-irrelevant.

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

$$A \simeq B \quad \triangleq \quad \Sigma f : A \to B.\, \mathsf{IsEquiv}(f)$$

Lemma 3.2. *Any isomorphism $f : A \to B$ is also an equivalence.*

The data of an equivalence $e : A \simeq B$ thus include two *transport functions*, denoted respectively $\uparrow_e : A \to B$ and $\downarrow_e : B \to A$. 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 3.3 (Univalent universe). A universe $\mathcal{U}$ is univalent if for any two types A and B in $\mathcal{U}$, the canonical map $A = B \to A \simeq B$ 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 Rocq/Coq interactive theorem prover [12]. Some more recent variants of dependent type theory [7, 19] feature a built-in computational univalence principle. They are used to implement experimental interactive theorem provers, such as Cubical Agda [59]. 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 $P : \square \to \square$, 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 [49].

Going back to our example from § 3.1, transferring along an equivalence $\mathbb{N} \simeq \mathsf{N}$ thus produces W_2 from V_1. Assuming univalence, one may achieve the more informative transport from V_1 to W_1, using *univalent parametricity* [53], which we briefly recall in the next section.

3.3 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.* [13, 14, 36] combine in what we refer to as the *raw parametricity translation*, which essentially defines inductively a logical relation $[\![T]\!]$ 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 [48]:

Theorem 3.4. *If $\Gamma \vdash t : T$ holds then $[\![\Gamma]\!] \vdash t : T$, $[\![\Gamma]\!] \vdash t' : T'$ and $[\![\Gamma]\!] \vdash [\![t]\!] : [\![T]\!]\ t\ t'$hold.*

Proof. By structural induction on the typing judgment, see for instance [36]. □

A key, albeit mundane ingredient of Theorem 3.4 is the fact that the rules of Figure 1 ensure that:

$$\vdash [\![\square_i]\!] \ : \ [\![\square_{i+1}]\!]\ \square_i\ \square_i \tag{8}$$

This translation generates precisely the statements expected from a parametric type family or program. For instance, the translation of a Π-type, given by Equation 7, is a type of relations on functions that relate those producing related outputs from related inputs. Concrete implementations of this translation are available [36, 54]; they generate and prove parametricity properties for type families or for constants, improved induction schemes, etc.

- Context translation:

$$[\![\langle\rangle]\!] = \langle\rangle \tag{1}$$

$$[\![\Gamma, x : A]\!] = [\![\Gamma]\!], x : A, x' : A', x_R : [\![A]\!]\ x\ x' \tag{2}$$

- Term translation:

$$[\![\square_i]\!] = \lambda A\, B.\, A \to B \to \square_i \tag{3}$$

$$[\![x]\!] = x_R \tag{4}$$

$$[\![t\ u]\!] = [\![t]\!]\ u\ u'\ [\![u]\!] \tag{5}$$

$$[\![\lambda x : A.\, t]\!] = \lambda (x : A)(x' : A')(x_R : [\![A]\!]\ x\ x').\, [\![t]\!] \tag{6}$$

$$[\![\Pi x : A.\, B]\!] = \lambda f\, f'.\, \Pi (x : A)(x' : A')(x_R : [\![A]\!]\ x\ x').\, [\![B]\!]\ (f\ x)\ (f'\ x') \tag{7}$$

Fig. 1. Raw parametricity translation for CC_ω.

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: the *univalent* parametricity translation of a universe of level i is the term denoted $\circledcirc_i$ and defined as:

$$[\![\square_i]\!]_u \triangleq \circledcirc_i \triangleq \lambda A\ B.\ \Sigma (R : A \to B \to \square_i)(e : A \simeq B).\Pi (a : A)(b : B).\, R\ a\ b \simeq (a =\downarrow_e b) \tag{9}$$

where $[\![\cdot]\!]_u$ now refers to the *univalent* parametricity translation. For any two types A and B, $[\![\square_i]\!]_u\ A\ B$ is a triple that packages a relation R, an equivalence e and a proof that R is equivalent to the functional relation associated with $\downarrow_e$. In fact, the following theorem holds [53]:

Theorem 3.5. *Assuming the univalence axiom, $[\![\square_i]\!]_u$ is equivalent to type equivalence, that is, for any two types A and B:*

$$\circledcirc_i\ A\ B \simeq (A \simeq B).$$

This observation is actually an instance of a more general technique for constructing syntactic models of type theory [18], based on attaching extra intensional specifications to negative type constructors. In these models, a standard way to recover the abstraction theorem consists in 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 $[\, T \,]$, is a dependent triple, which equips a relation with the additional data prescribed by the interpretation $[\![\square_i]\!]_u$ of the universe. The translation $[\![T]\!]_u$ of T as a *type* is the relation itself, that is, the first projection of the dependent triple $[\, T \,]$ onto its first component, denoted $\mathrm{rel}([\, T \,])$. The second component of the triple is an equivalence, and the third one is a coherence data between the first and the second one. For the sake of convenience, Figure 2 recalls the definition of the univalent parametricity translation, as provided in its original presentation [53, Figure 4]. Our figure uses similar notations as the original presentation, in particular, dependent triples are denoted as $(\_;\_;\_)$, $id_{\square_i}$ is the identity equivalence on universe $\square_i$ and $Equiv_\Pi$ constructs an equivalence between types $\Pi x : A.B$ and $\Pi x : A'.B'$ from its arguments. Observe that one recovers the raw parametricity translation by projecting every triple onto its relation, except in the case of $\square_i$ where there relation part is fundamentally different. The crux of this translation lies in the construction of terms $univ_{\square_i}$ and $univ_\Pi$, which provide the coherence data needed to construct the triple translating a universe and a Π type, respectively.

Theorem 3.6 states the abstraction theorem of the univalent parametricity translation [53], where $\vdash_u$ denotes a typing judgment of CC_ω assuming univalence:

- Context translation:

$$[\![\langle\rangle]\!]_u = \langle\rangle \tag{10}$$

$$[\![\Gamma, x : A]\!]_u = [\![\Gamma]\!]_u, x : A, x' : A', x_R : [\![A]\!]_u\ x\ x' \tag{11}$$

- Term translation:

$$[\ \square_i\ ] = p_{\square_i} \tag{12}$$

$$[\ x\ ] = x_R \tag{13}$$

$$[\ t\ u\ ] = [\ t\ ]\ u\ u'\ [\ u\ ] \tag{14}$$

$$[\ \lambda x : A.\ t\ ] = \lambda (x : A)(x' : A')(x_R : [\![A]\!]_u\ x\ x').\ [\ t\ ] \tag{15}$$

$$[\ \Pi x : A.\ B\ ] = p_\Pi\ [\ A\ ]\ [\ B\ ] \tag{16}$$

- Type translation:

$$[\![A]\!]_u = \mathsf{rel}([\ A\ ])$$

- With:

$$\begin{aligned} p_{\square_i} =& (\circledcirc_i; id_{\square_i}; univ_{\square_i}) \\ p_\Pi\ A_R\ B_R =& (\lambda f f'.\ \Pi (x : A)(x' : A')(x_R : \mathsf{rel}(A_R)\ x\ x').\ \mathsf{rel}(B_R)\ (f\ x)(f'\ x'); \\ & univ_\Pi\ A_R\ B_R; univ_\Pi\ A_R\ B_R) \end{aligned}$$

  omitting implicit arguments.

Fig. 2. Univalent parametricity translation for CC_ω.

Theorem 3.6. *If* $\Gamma \vdash t : T$ *then* $[\![\Gamma]\!]_u \vdash_u [\ t\ ] : [\![T]\!]_u\ t\ t'$.

Note that proving this theorem involves in particular ensuring that:

$$\vdash_u [\ \square_i\ ] : [\![\square_{i+1}]\!]_u\ \square_i\ \square_i \quad \text{and} \quad \mathsf{rel}([\ \square_i\ ]) \equiv [\![\square_i]\!]_u. \tag{17}$$

The univalence axiom makes this possible by allowing to define

$$\begin{aligned} \vdash_u univ_{\square_i} &: \Pi A\ B : \square_i.\ \circledcirc_i\ A\ B \simeq (A = B) \\ \vdash_u univ_\Pi\ A_R B_R &: \Pi (f : \Pi x : A.\ B)(f' : \Pi x : A'.\ B'). \\ & \big(\Pi (x : A)(x' : A')(x_R : \mathsf{rel}(A_R)\ x\ x').\ \mathsf{rel}(B_R)\ (f\ x)(f'\ x')\big) \\ & \simeq (f =_{\downarrow Equiv_\Pi A_R B_R}\ g). \end{aligned}$$

Indeed, assuming the univalence axiom, $univ_{\square_i}$ is a corollary of Theorem 3.5, and the definition of term $univ_\Pi$ follows from dependent functional extensionality, which is also a consequence of the univalence axiom. Importantly, this univalent parametricity translation can be seamlessly extended so as to also make use of a global context of user-defined equivalences.

However, because of the interpretation of universes given by Equation 9, 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. Moreover, the definition of terms $univ_{\square_i}$ and $univ_\Pi$ crucially rely on the univalence principle. Therefore, an execution of the translation which involves cases 12 and 16 may incur unnecessary dependencies on the univalence axiom in the propagation of type equivalences. This problem is illustrated in Example 2.2.

4 An operational view on parametricity translations

We propose here a sequent style presentation of translations, which fits closely the type system of CC_ω, while explaining in a consistent way the transformations of terms $\mathcal{T}_{CC_\omega}$ and contexts $\mathcal{C}_{CC_\omega}$. 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 3.3).

4.1 Raw parametricity sequents

We introduce *parametricity contexts* $\mathcal{R}_{CC_\omega}$, 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:

$$\mathcal{R}_{CC_\omega} \ni \Xi ::= \langle\rangle \mid \Xi,\ x \sim x' \because x_R$$

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

$$\mathrm{Var}(\langle\rangle) = \langle\rangle \qquad \mathrm{Var}(\Xi,\ x \sim x' \because x_R) = \mathrm{Var}(\Xi), x, x', x_R.$$

A parametricity context Ξ is *well-formed*, written $\Xi \vdash$, if the sequence $\mathrm{Var}(\Xi)$ 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 t, t', t_R of CC_ω. Parametricity judgments, denoted as:

$$\Xi \vdash t \sim t' \because t_R,$$

are defined by rules of Figure 3 and read *in context Ξ, term t translates to the term t', because t_R.*

$$\frac{}{\Xi \vdash \square_i \sim \square_i \because \lambda(A\,B : \square_i).\, A \to B \to \square_i}\ (\textsc{ParamSort}) \qquad \frac{(x, x', x_R) \in \Xi \qquad \Xi \vdash}{\Xi \vdash x \sim x' \because x_R}\ (\textsc{ParamVar})$$

$$\frac{\Xi \vdash t \sim t' \because t_R \quad \Xi \vdash u \sim u' \because u_R}{\Xi \vdash t\ u \sim t'\ u' \because t_R\ u\ u'\ u_R}\ (\textsc{ParamApp})$$

$$\frac{\begin{array}{c} x, x' \notin \mathrm{Var}(\Xi) \\ \Xi \vdash A \sim A' \because A_R \qquad \Xi, x \sim x' \because x_R \vdash t \sim t' \because t_R \end{array}}{\Xi \vdash \lambda x : A.\, t \sim \lambda x' : A'.\, t' \because \lambda x\, x'\, x_R.\, t_R}\ (\textsc{ParamLam})$$

$$\frac{\begin{array}{c} x, x' \notin \mathrm{Var}(\Xi) \\ \Xi \vdash A \sim A' \because A_R \qquad \Xi, x \sim x' \because x_R \vdash B \sim B' \because B_R \end{array}}{\Xi \vdash \Pi x : A.\, B \sim \Pi x' : A'.\, B' \because \lambda f\, g.\, \Pi x\, x'\, x_R.\, B_R\ (f\ x)\ (g\ x')}\ (\textsc{ParamPi})$$

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

Lemma 4.1. *The relation associating a term t with pairs (t', t_R) such that $\Xi \vdash t \sim t' \because t_R$ holds, with Ξ a well-formed parametricity context, is* functional. *More precisely, for any well-formed parametricity*

context Ξ*:*

$$\forall t, t', u', t_R, u_R, \quad \Xi \vdash t \sim t' \mathrel{\because} t_R \quad \wedge \quad \Xi \vdash t \sim u' \mathrel{\because} u_R \\ \implies (t', t_R) = (u', u_R)$$

Proof. Immediate by induction on the syntax of t. □

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

Definition 4.2. A parametricity context Ξ is *admissible* for a well-formed typing context Γ, denoted $\Gamma \triangleright \Xi$, when Ξ is well-formed as a parametricity context and Γ provides coherent type annotations for all terms in Ξ, that is,

$$\frac{}{\langle\rangle \triangleright \langle\rangle}\ {\scriptstyle \triangleright\langle\rangle} \qquad \frac{\Gamma \triangleright \Xi}{(\Gamma, x : A, x' : A', x_R : A_R\ x\ x') \triangleright (\Xi, x \sim x' \mathrel{\because} x_R)}\ {\scriptstyle \triangleright,\cdot}$$

We can now state and prove an abstraction theorem:

Theorem 4.3 (Abstraction theorem).

$$\frac{\Gamma \triangleright \Xi \qquad \Gamma \vdash t : A \qquad \Xi \vdash t \sim t' \mathrel{\because} t_R \qquad \Xi \vdash A \sim A' \mathrel{\because} A_R}{\Gamma \vdash t' : A' \qquad \textit{and} \qquad \Gamma \vdash t_R : A_R\ t\ t'}$$

Proof. By induction on the derivation of $\Xi \vdash t \sim t' \mathrel{\because} t_R$. □

Constants. Postulating arbitrary axioms, stating the existence of parametricity translations for some constants, easily break the abstraction theorem. We refer to Tabareau et al. [53] for a discussion about the unfortunate interaction of conversion and elimination rules for the types `𝕟` and `N` of Example 2.1. Giving up on δ-expansion, it is however possible to extend the translation to inert, opaque constants in the following way. Given:

- a set $\mathcal{C}$ of symbols;
- for each $c \in \mathcal{C}$, a term $T_c \in \mathcal{T}_{CC_\omega}$, such that $\vdash T_c : \square$;
- a partial translation function $\xi : \mathcal{C} \to \mathcal{C}^2$ such that for all c, c' and c_R with $\xi(c) = (c', c_R)$, we have

$$\vdash T_c \sim T_{c'} \mathrel{\because} T_{c_R} \text{ and } \vdash c_R : T_{c_R}\ c\ c'$$

the rules of Figure 4 shall respectively extend those of CC_ω and of Param.

$$\frac{c \in \mathcal{C}}{\vdash c : T_c}\ (\textsc{Const}) \qquad \frac{c \in \mathcal{C} \qquad \xi(c) = (c', c_R)}{\vdash c \sim c' \mathrel{\because} c_R}\ (\textsc{ParamConst})$$

Fig. 4. Extension of CC_ω and its raw parametricity translation with constants.

The abstraction theorem still holds, as the case of neutral constants can be treated exactly as that of variables.

We now show how to use raw parametricity to deal with Example 2.1. For the sake of conciseness, we provide only the translation of one side of the inequality. Reusing notations from Section 1, we pose

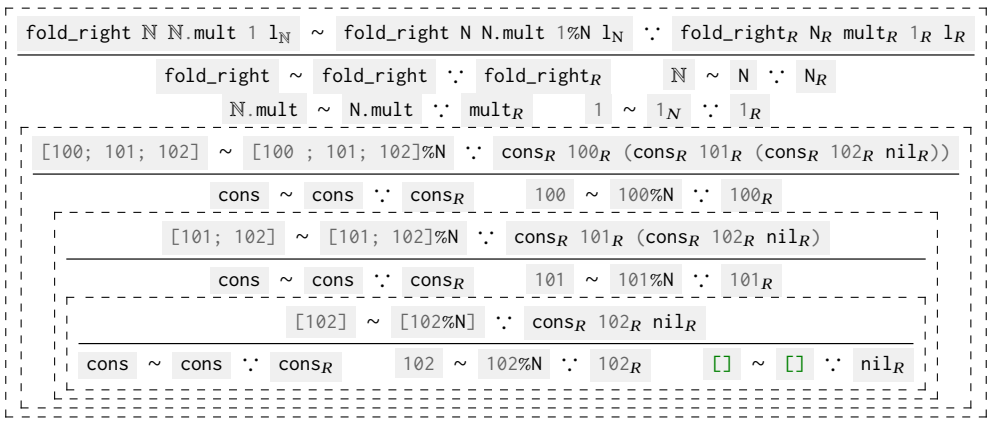
$$\mathsf{N}_R\ m\ n \triangleq (\downarrow_{\mathbb{N}} m = n)$$

and we have `1_R : N_R 1 1%N`, `100_R : N_R 100 100%N`, `101_R : N_R 101 101%N`, and `102_R : N_R 102 102%N`, which can all be obtained by computation within Rocq/Coq. We furthermore assume a proof

$$\mathrm{mult}_R : \forall m_1 n_1, \mathsf{N}_R\ m_1\ n_1 \implies \forall m_2 n_2, \mathsf{N}_R\ m_2\ n_2 \implies \mathsf{N}_R\ (m_1 \times_{\mathbb{N}} m_2)\ (n_1 \times_N n_2)$$

which can be read as a proof of multiplicative morphism between N and $\mathbb{N}$.

Let us represent the trace of the execution of the logic program computing the translation `fold_right ℕ.mult ℕ 1 1_ℕ ~ t' ⋱ t_R`, which synthesizes both t' and t_R, through a top-down derivation tree. We write rules in the reverse order compared to the presentation of the calculus from Figure 1. Since the tree would widen out of the page, we use dashed-boxes to delimit non-leaf subtrees of the derivation.



Finally, by the abstraction theorem, we get that `fold_right_R N_R mult_R 1_R 1_R` is a proof of the following: `N_R (fold_right ℕ ℕ.mult 1 1_ℕ) (fold_right N N.mult 1%N 1_N)`, which by definition of `N_R`, means `↓_ℕ (fold_right ℕ ℕ.mult 1 1_ℕ) = fold_right N N.mul 1%N 1_N`. This works well enough for closed statements about values in a given datatype, such at `ℕ` or `bool`, but falls short of scaling to CC_ω.

4.2 Univalent parametricity sequents

We now propose in Figure 5 a rephrased version of the univalent parametricity translation defined on Figure 2, using the same sequent style. Its parametricity judgments are denoted:

$$\Xi \vdash_u t \sim t' \mathrel{⋱} t_R$$

where Ξ is a parametricity context and t, t', and t_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.

Theorem 4.4 (Univalent abstraction theorem).

$$\frac{\Gamma \triangleright \Xi \qquad \Gamma \vdash t : A \qquad \Xi \vdash_u t \sim t' \mathrel{⋱} t_R \qquad \Xi \vdash_u A \sim A' \mathrel{⋱} A_R}{\Gamma \vdash t' : A' \qquad \textit{and} \qquad \Xi \vdash_u t_R : \mathit{rel}(A_R)\ t\ t'}$$

$$\frac{}{\Xi \vdash_u \square_i \sim \square_i \mathrel{\because} p_{\square_i}}\ (\textsc{UParamSort}) \qquad \frac{(x, x', x_R) \in \Xi \qquad \Xi \vdash}{\Xi \vdash_u x \sim x' \mathrel{\because} x_R}\ (\textsc{UParamVar})$$

$$\frac{\Xi \vdash_u t \sim t' \mathrel{\because} t_R \qquad \Xi \vdash_u u \sim u' \mathrel{\because} u_R}{\Xi \vdash_u t\ u \sim t'\ u' \mathrel{\because} t_R\ u\ u'\ u_R}\ (\textsc{UParamApp})$$

$$\frac{\begin{array}{c} x, x' \notin \mathrm{Var}(\Xi) \\ \Xi \vdash_u A \sim A' \mathrel{\because} A_R \qquad \Xi, x \sim x' \mathrel{\because} x_R \vdash_u t \sim t' \mathrel{\because} t_R \end{array}}{\Xi \vdash_u \lambda x : A.\, t \sim \lambda x' : A'.\, t' \mathrel{\because} \lambda x\, x'\, x_R.\, t_R}\ (\textsc{UParamLam})$$

$$\frac{\begin{array}{c} x, x' \notin \mathrm{Var}(\Xi) \\ \Xi \vdash_u A \sim A' \mathrel{\because} A_R \qquad \Xi, x \sim x' \mathrel{\because} x_R \vdash_u B \sim B' \mathrel{\because} B_R \end{array}}{\Xi \vdash_u \Pi x : A.\, B \sim \Pi x' : A'.\, B' \mathrel{\because} p_\Pi\ A_R\ B_R}\ (\textsc{UParamPi})$$

Fig. 5. UParam: univalent parametricity rules

Proof. By induction on the derivation of $\Xi \vdash_u t \sim t' \mathrel{\because} t_R$. □

Remark 4.5. In Theorem 4.4, $\mathsf{rel}(A_R)$ is a term of type $A \to A' \to \square$. Indeed:

$$\frac{\Gamma \vdash_u A : \square_i \qquad \Xi \vdash_u A \sim A' \mathrel{\because} A_R \qquad \Gamma \rhd \Xi}{\Gamma \vdash_u A_R : \mathsf{rel}(p_{\square_i})\ A\ A'}$$

entails A_R has type

$$\begin{aligned} \mathsf{rel}(p_{\square_i})\ A\ A' \ &\equiv\ \circledcirc_i\ A\ A' \\ &\triangleq \Sigma(R : A \to B \to \square_i)(e : A \simeq B).\Pi(a : A)(b : B).\, R\ a\ b \simeq (a = \downarrow_e b). \end{aligned}$$

Constants. Like in the case of raw parametricity, the univalent parametricity translation can be extended with neutral, opaque constants in the following way. Given,

- a set $\mathcal{C}$ of symbols;
- for each $c \in \mathcal{C}$, a term $T_c \in \mathcal{T}_{CC_\omega}$, such that $\vdash_u T_c : \square$;
- a partial translation function $\xi : \mathcal{C} \to \mathcal{C}^2$ such that for all c, c' and c_R with $\xi(c) = (c', c_R)$, we have

$$\vdash_u T_c \sim T_{c'} \mathrel{\because} T_{c_R} \text{ and } \vdash_u c_R : \mathsf{rel}(T_{c_R})\ c\ c'$$

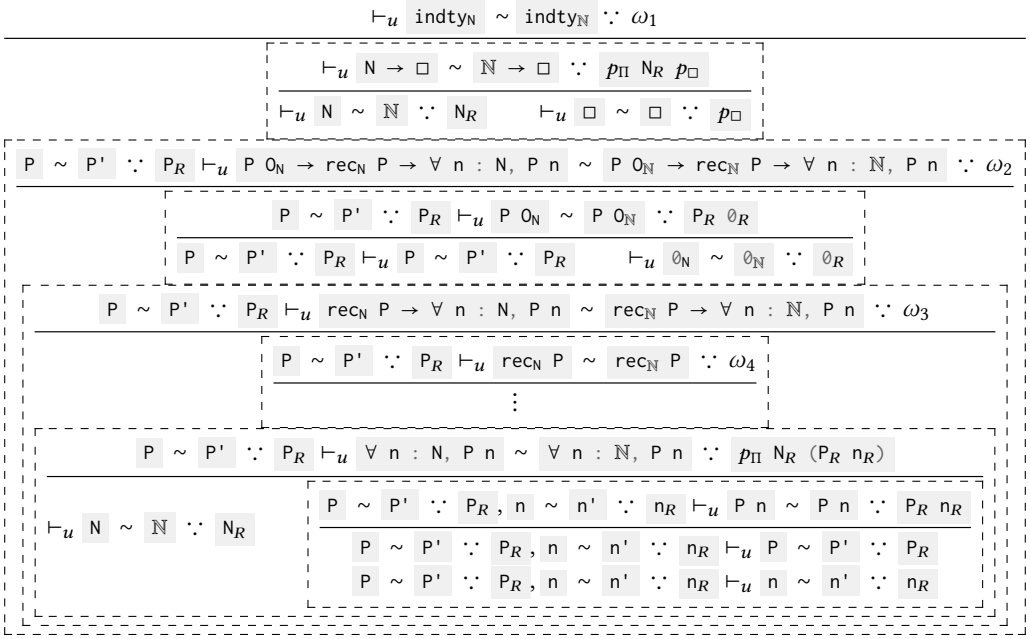
the rules of Figure 6 shall respectively extend those of CC_ω and UParam.

$$\frac{c \in \mathcal{C}}{\vdash_u c : T_c}\ (\textsc{Const}) \qquad \frac{c \in \mathcal{C} \qquad \xi(c) = (c', c_R)}{\vdash_u c \sim c' \mathrel{\because} c_R}\ (\textsc{UParamConst})$$

Fig. 6. Additional constant rules for raw parametricity

The abstraction theorem still holds, as the case of neutral constants can be treated exactly as that of variables.

We show how to use univalent parametricity to deal with Example 2.2. We introduce abbreviations for the types of the induction principles `indty_N := ∀ P : N → □, P 0_N → rec_N P → ∀ n : N, P n` and `indty_ℕ := ∀ P : ℕ → □, P 0_ℕ → rec_ℕ P → ∀ n : ℕ, P n` where `rec_N P := ∀ n : N, P n → P (S_N n)` and `rec_ℕ P := ∀ n : ℕ, P n → P (S_ℕ n)` for the sake of conciseness.



where

$\omega_1 =$ `p_Π (p_Π N_R p_□) ω2`

$\omega_2 =$ `p_Π (P_R 0_R) ω3`

$\omega_3 =$ `p_Π ω4 (p_Π N_R (P_R n_R))`

ω_4 is the result of the omitted subtree

By the abstraction theorem, ω_1 eventually has type

```
⊡ (∀ P : N → □, P 0_N → rec_N P → ∀ n : N, P n) (∀ P : ℕ → □, P 0_ℕ → rec_ℕ P → ∀ n : ℕ, P n)
```

Which means the two terms are equivalent as elements of □.

Remark 4.6. If we simply want to replace one goal with the other, proving their equivalence is too strong: we just need an implication.

Remark 4.7. Term ω_1 requires univalence, because it features $p_\square$ as a subterm.

5 Type equivalence in kit form

In this section, we propose (§ 5.1) an equivalent, modular presentation of type equivalence, phrased as a nested sigma type. Then (§ 5.2), we carve a hierarchy of structures on relations out of this dependent tuple, selectively picking pieces, in order to revisit parametricity translations through the lens of this finer grained analysis of the relational interpretations of types in § 6.1.

We make heavy use of the axiom of univalence to prove the lemmas and theorems of this section, and we formalized them thoroughly in the companion material. However, none of these proofs are used in the code generated by the plugin. We comment in § 7 on the possible use of axioms in generated code.

5.1 Disassembling type equivalence

Let us start by observing that Definition 3.1, 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 elided in Definition 3.1, is actually:

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

where $\mathrm{ap}_f(t)$ 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 [57] however suggests a symmetrical definition of type equivalence, via functional relations.

Definition 5.1. A relation $R : A \to B \to \square_i$, is *functional* when:

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

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

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

Lemma 5.2. *For any types $A, B : \square$, type $A \simeq B$ is equivalent to:*

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

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

We sketch below a proof of this result, left as an exercise in [57]. The essential argument is the following characterization of functional relations:

Lemma 5.3. *The type of functions is equivalent to the type of functional relations; i.e., for any types $A, B : \square$, we have $(A \to B) \simeq \Sigma R : A \to B \to \square.\ \mathit{IsFun}(R)$.*

Proof. The proof goes by chaining the following equivalences:

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

□

Proof of Lemma 5.2. The proof goes by chaining the following equivalences, where the type of f is always $A \to B$ and the type of R is $A \to B \to \square$:

$$\begin{aligned}
(A \simeq B) &\simeq \Sigma f : A \to B.\, \mathsf{IsEquiv}(f) && \text{by definition of } (A \simeq B)\\
&\simeq \Sigma f.\, \Pi b : B.\, \mathsf{IsContr}(\Sigma a.f\ a = b) && \text{standard result in HoTT}\\
&\simeq \Sigma f.\, \mathsf{IsFun}(\lambda(b : B)(a : A).\, f\ a = b) && \text{by definition of } \mathsf{IsFun}(\cdot)\\
&\simeq \Sigma\left(\varphi : \Sigma R.\, \mathsf{IsFun}(R)\right).\, \mathsf{IsFun}(\pi_1(\varphi)^{-1}) && \text{by Lemma 5.3}\\
&\simeq \Sigma R.\, \mathsf{IsFun}(R) \times \mathsf{IsFun}(R^{-1}) && \text{by associativity of } \Sigma.
\end{aligned}$$

□

However, the definition of type equivalence provided by Lemma 5.2 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 5.4. For any types $A, B : \square$, a relation $R : A \to B \to \square$, is a *univalent map*, denoted $\mathsf{IsUmap}(R)$ when there exists a function $m : A \to B$ together with:

$$\begin{aligned}
g_1 &: \Pi(a : A)(b : B).\, m\ a = b \to R\ a\ b\\
\text{and}\quad g_2 &: \Pi(a : A)(b : B).\, R\ a\ b \to m\ a = b\\
\text{such that}\quad &\Pi(a : A)(b : B).\, (g_1\ a\ b) \circ (g_2\ a\ b) \doteqdot id.
\end{aligned}$$

Now comes the crux lemma of this section.

Lemma 5.5. *For any types $A, B : \square$ and any relation $R : A \to B \to \square$*

$$\mathit{IsFun}(R) \simeq \mathit{IsUmap}(R).$$

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

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

where $t =_e u$ denotes the equality of the transport of term t along equality e with term u. After a suitable reorganization of the sigma types we are left to show that

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

whose proof we do not detail, referring the reader to the corresponding code. □

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

Theorem 5.6. *For any types $A, B : \square_i$, we have:*

$$(A \simeq B) \simeq \boxdot_i^{\top}\ A\ B$$

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

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

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.

5.2 Reassembling type equivalence

Definition 5.4 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 5.7. For $n, k \in \{0, 1, 2_{\mathrm{a}}, 2_{\mathrm{b}}, 3, 4\}$, and $\alpha = (n, k)$, relation $\boxdot^{\alpha} : \square \to \square \to \square$, is defined as:

$$\boxdot^{\alpha} \triangleq \lambda(A\ B : \square).\Sigma(R : A \to B \to \square).\mathsf{Class}_{\alpha}\ R$$

where the *map class* $\mathsf{Class}_{\alpha}\ R$ itself unfolds to a pair type $(\mathsf{M}_n\ R) \times (\mathsf{M}_k\ R^{-1})$, with M_i defined as:[2]

$$\begin{aligned}
&\mathsf{M}_0\ R \triangleq .\\
&\mathsf{M}_1\ R \triangleq (A \to B)\\
&\mathsf{M}_{2_{\mathrm{a}}}\ R \triangleq \Sigma m : A \to B.\ G_{2_{\mathrm{a}}}\ m\ R \quad \text{with } G_{2_{\mathrm{a}}}\ m\ R \triangleq \Pi a\, b.\ m\ a = b \to R\ a\ b\\
&\mathsf{M}_{2_{\mathrm{b}}}\ R \triangleq \Sigma m : A \to B.\ G_{2_{\mathrm{b}}}\ m\ R \quad \text{with } G_{2_{\mathrm{b}}}\ m\ R \triangleq \Pi a\, b.\ R\ a\ b \to m\ a = b\\
&\mathsf{M}_3\ R \triangleq \Sigma m : A \to B.\ (G_{2_{\mathrm{a}}}\ m\ R) \times (G_{2_{\mathrm{b}}}\ m\ R)\\
&\mathsf{M}_4\ R \triangleq \Sigma m : A \to B.\ \Sigma(g_1 : G_{2_{\mathrm{a}}}\ m\ R).\ \Sigma(g_2 : G_{2_{\mathrm{b}}}\ m\ R).\ \Pi a\, b.\ (g_1\ a\ b) \circ (g_2\ a\ b) \doteqdot \mathit{id}
\end{aligned}$$

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

Definition 5.8. We denote $\mathcal{A}$ the set $\{0, 1, 2_{\mathrm{a}}, 2_{\mathrm{b}}, 3, 4\}^2$, used to index map classes in Definition 5.7. 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 5.9. Relation $\boxdot^{(4,4)}$ of Definition 5.7 indeed coincides with the relation $\boxdot^{\top}$ introduced in Theorem 5.6. It is equivalent to $\boxdot$ by Theorem 3.5. Similarly, we denote $\boxdot^{\bot}$ the relation $\boxdot^{(0,0)}$, for which we obviously have $\boxdot_i^{\bot}\ A\ B \simeq (A \to B \to \square_i)$.

Remark 5.10. Here is a useful dictionary for specific points in the hierarchy:

Functions. For r of type $\boxdot^{(1,0)}\ A\ B$, $\mathsf{map}(r)$ is an arbitrary function $f : A \to B$, unrelated to relation $\mathsf{rel}(r)$. This can for instance be used to synthesize the proof of an implication, or the existence of a coercion from type A to type B.

Univalent maps. For r of type $\boxdot^{(4,0)}\ A\ B$, $\mathsf{rel}(r)$ is a univalent map, in the sense of Definition 5.4.

[2]For the sake of readability, we omit implicit arguments, *e.g.*, although M_i has type $\lambda(S\ T : \square).\ (S \to T \to \square) \to \square$, we write $\mathsf{M}_n\ R$ for $(\mathsf{M}_n\ A\ B\ R)$.

Retractions. For r of type $\boxdot^{(4,2_\mathrm{a})}\ A\ B$, $\mathsf{rel}(r)$ is the graph of a retraction (*i.e.*, a surjective univalent map with an explicit partial left inverse) of type $A \to B$. This is the nature of relation at stake in Example 2.4.

Sections. For r of type $\boxdot^{(4,2_\mathrm{b})}\ A\ B$, $\mathsf{rel}(r)$ is the graph of a section (*i.e.*, an injective univalent map with explicit partial right inverse) of type $A \to B$. This is the nature of relation at stake in Example 2.3.

Equivalences. For r of type $\boxdot^{(3,3)}$, r is an isomorphism between A and B, and for r of type $\boxdot^{(4,4)}$, r is an equivalence between A and B. As isomorphisms can always be promoted to a type equivalences, it is always possible to construct a term of type $\boxdot^{(4,4)}$ from a term of type $\boxdot^{(3,3)}$. This is the nature of relation at stake in Examples 2.1 and 2.2.

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.

6 A calculus for proof transfer

This section introduces Trocq, a framework for proof transfer designed as a generalization of both raw and univalent parametricity translations, so as to allow for interpreting types as instances of the structures introduced in Section 5.2. In this framework, the translation of a term may vary, depending on the strength of the expected result. For instance, we may expect to obtain a univalent translation, or a raw translation, and more generally, we may request a witness of any of the nature of relations introduced in § 5. We thus need to inform the translation with classes from $\mathcal{A}$, so as to guide the synthesis of the output term from the input one.

Therefore, we first explain how to generalize the constants $p_{\square_i}$ and p_Π (from § 4.2) according to those classes (§ 6.1). Then we introduce CC^+_ω (§ 6.2), a Calculus of Constructions with annotations on sorts and subtyping which reflect lattice $\mathcal{A}$, so as to guide the translation. We finally define (§ 6.3) the Trocq calculus itself.

6.1 Populating the hierarchy of relations

We shall now revisit the parametricity translations of Section 3.3. In particular, recall that Theorem 5.6, together with Equation 17, relies on the existence of a term $p_{\square_i}$ such that:

$$\vdash_u p_{\square_i} : \boxdot^\top_{i+1}\ \square_i\ \square_i \quad \text{and} \quad \mathsf{rel}(p_{\square_i}) \simeq \boxdot^\top_i.$$

Otherwise said, the relation $\boxdot^\top : \square \to \square \to \square$ can be endowed with a $\boxdot^\top$ structure, assuming univalence. Similarly, Equation 8, for the raw parametricity translation, can be read as the fact that relation $\boxdot^\bot$ on universes can be endowed with a $\boxdot^\bot\ \square\ \square$ structure.

Now the hierarchy of structures introduced by Definition 5.7 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 $\boxdot^\alpha : \square \to \square \to \square$ 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 $p^{\alpha,\beta}_\square$ such that:

$$\vdash_u p^{\alpha,\beta}_\square : \boxdot^\beta\ \square\ \square \quad \text{and} \quad \mathsf{rel}(p^{\alpha,\beta}_\square) \equiv \boxdot^\alpha \tag{18}$$

Note that we aim here at a definitional equality between $\mathsf{rel}(p^{\alpha,\beta}_\square)$ and $\boxdot^\alpha$, rather than at an equivalence. It is easy to see that a term $p^{\alpha,\bot}_\square$ exists for any $\alpha \in \mathcal{A}$, as $\boxdot^\bot$ requires no structure on

the relation. On the other hand, it is not possible to construct a term $p_{\square}^{\bot,\top}$, *i.e.,* to turn an arbitrary relation into a type equivalence.

Definition 6.1. We denote $\mathcal{D}_{\square}$ the following subset of $\mathcal{A}^2$:

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

The corresponding code 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 $\beta \notin \{0, 1, 2_{\mathrm{a}}\}^2$.

We now turn to the other essential ingredient of the abstraction theorem, and investigate the possible structures $\circledcirc^{\gamma}$ endowing the relational interpretation of a product type $\Pi x : A.\, B$, given relational interpretations for types A and B respectively equipped with structures $\circledcirc^{\alpha}$ and $\circledcirc^{\beta}$.

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_{Π}^{γ} such that the following statements both hold:

$$\frac{\Gamma \vdash A_R \;:\; \circledcirc^{\alpha}\; A\; A' \qquad \Gamma,\; x : A,\; x' : A',\; x_R : A_R\; x\; x' \;\vdash\; B_R \;:\; \circledcirc^{\beta}\; B\; B'}{\Gamma \;\vdash\; p_{\Pi}^{\gamma}\; A_R\; B_R \;:\; \circledcirc^{\gamma}\; (\Pi x : A.\, B)\; (\Pi x' : A'.\, B')}$$

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

The corresponding collection of triples can actually be described as a function $\mathcal{D}_{\Pi} : \mathcal{A} \to \mathcal{A}^2$, such that $\mathcal{D}_{\Pi}(\gamma) = (\alpha, \beta)$ provides sufficient requirements on the structures associated with A and B, with respect to the partial order on $\mathcal{A}^2$. The corresponding code 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 the annotations $m_A, n_A, m_B, n_B \in \{0, 1, 2_{\mathrm{a}}, 2_{\mathrm{b}}, 3, 4\}$ 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 7 the values of $\mathcal{D}_{\Pi}(m, 0)$.

m	$\mathcal{D}_{\Pi}(m,0)_1$	$\mathcal{D}_{\Pi}(m,0)_2$
0	$(0,0)$	$(0,0)$
1	$(0,2_{\mathrm{a}})$	$(1,0)$
2_{a}	$(0,4)$	$(2_{\mathrm{a}},0)$
2_{b}	$(0,2_{\mathrm{a}})$	$(2_{\mathrm{b}},0)$
3	$(0,4)$	$(3,0)$
4	$(0,4)$	$(4,0)$

m	$\mathcal{D}_{\to}(m,0)_1$	$\mathcal{D}_{\to}(m,0)_2$
0	$(0,0)$	$(0,0)$
1	$(0,1)$	$(1,0)$
2_{a}	$(0,2_{\mathrm{b}})$	$(2_{\mathrm{a}},0)$
2_{b}	$(0,2_{\mathrm{a}})$	$(2_{\mathrm{b}},0)$
3	$(0,3)$	$(3,0)$
4	$(0,4)$	$(4,0)$

Fig. 7. Constraints for product and arrow types

Note that in the case of a non-dependent product, constructing $p_{\to}^{\gamma}$ requires less structure on the domain A of an arrow type $A \to B$, which motivates the introduction of function $\mathcal{D}_{\to}(\gamma)$. Using the combinator for dependent products to interpret the special case of an arrow type, albeit correct,

potentially pulls in unnecessary structure (and axiom) requirements. The corresponding code thus includes a construction of terms $p^{\gamma}_{\rightarrow}$ for any $\gamma \in \mathcal{A}$.

The two tables in Fig.7 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. We conjecture that the dependencies given in Figure 7 are minimal.

6.2 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 5.2 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 $\square^{\alpha}$, then the associated relation A_R has type $\boxdot^{\alpha}\ A\ A'$. On concrete applications, where type A' is synthesized together with A_R, annotation α indicates the nature of parametricity translation expected by the user. For instance, if A has type $\square^{\bot}$, then the framework will compute the raw translation of A, and if A has type $\square^{\top}$, it will compute the univalent one. But the framework also offers a palette of other options, put to good use in the corresponding implementations to the examples of Section 2. The syntax of CC^{+}_{ω} is thus:

$$\mathcal{T}_{CC^{+}_{\omega}} \ni M, N, A, B ::= \square^{\alpha}_{i} \mid x \mid M\ N \mid \lambda x : A.\, M \mid \Pi x : A.\, B$$
$$\alpha \in \mathcal{A} = \{0, 1, 2_a, 2_b, 3, 4\}^2 \qquad i \in \mathbb{N}$$

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 $\mathrm{rel}(A_R)\ x\ x'$ 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 $\mathcal{D}_{\Pi}$, introduced in §6.1.

Contexts are defined as usual, but typing terms in CC^{+}_{ω} requires defining a *subtyping* relation $\preccurlyeq$, defined by the rules of Figure 8. The typing rules of CC^{+}_{ω} are available in Figure 9 and follow standard presentations [9]. 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 $B \in \mathcal{T}_{CC^{+}_{\omega}}$ are terms in CC^{+}_{ω} and $\Gamma \in \mathcal{C}_{CC^{+}_{\omega}}$ is a context in CC^{+}_{ω}.

We show that our extension is conservative over CC_{ω}, by defining an erasure function for terms $|\cdot|^{-} : \mathcal{T}_{CC^{+}_{\omega}} \rightarrow \mathcal{T}_{CC_{\omega}}$ and for contexts, defined in Figure 10.

We show that the erasure of subtyping is convertibility in CC_{ω}:

Lemma 6.2 (Conservativity of conversion in CC^{+}_{ω}).

$$\Gamma \vdash_{CC^{+}_{\omega}} A \preccurlyeq B \implies |\Gamma|^{-} \vdash_{CC_{\omega}} |A|^{-} \equiv |B|^{-}$$

Proof. By induction on the derivation. □

Finally we show that CC^{+}_{ω} is a conservative extension over CC_{ω}:

$$\frac{\Gamma \vdash_+ A : K \qquad \Gamma \vdash_+ B : K \qquad A \equiv B}{\Gamma \vdash_+ A \preccurlyeq B}\ (\textsc{SubConv}) \qquad \frac{\alpha \geq \beta \qquad i \leq j}{\Gamma \vdash_+ \square_i^\alpha \preccurlyeq \square_j^\beta}\ (\textsc{SubSort})$$

$$\frac{\Gamma \vdash_+ M'\ N : K \qquad \Gamma \vdash_+ M \preccurlyeq M'}{\Gamma \vdash_+ M\ N \preccurlyeq M'\ N}\ (\textsc{SubApp}) \qquad \frac{\Gamma, x : A \vdash_+ M \preccurlyeq M'}{\Gamma \vdash_+ \lambda x : A.\, M \preccurlyeq \lambda x : A.\, M'}\ (\textsc{SubLam})$$

$$\frac{\Gamma \vdash_+ \Pi x : A.\, B : \square_i \qquad \Gamma \vdash_+ A' \preccurlyeq A \qquad \Gamma, x : A' \vdash_+ B \preccurlyeq B'}{\Gamma \vdash_+ \Pi x : A.\, B \preccurlyeq \Pi x : A'.\, B'}\ (\textsc{SubPi}) \qquad K ::= \square_i \mid \Pi x : A.\, K$$

Fig. 8. Subtyping rules for CC_ω^+

$$\frac{\Gamma \vdash_+ M : A \qquad \Gamma \vdash_+ A \preccurlyeq B}{\Gamma \vdash_+ M : B}\ (\textsc{Conv}^+) \qquad \frac{(\alpha, \beta) \in \mathcal{D}_\square}{\Gamma \vdash_+ \square_i^\alpha : \square_{i+1}^\beta}\ (\textsc{Sort}^+)$$

$$\frac{(x, A) \in \Gamma \qquad \Gamma \vdash_+}{\Gamma \vdash_+ x : A}\ (\textsc{Var}^+) \qquad \frac{\Gamma \vdash_+ A : \square_i \qquad x \notin \mathrm{Var}(\Gamma)}{\Gamma,\ x : A \vdash_+}\ (\textsc{Context}^+)$$

$$\frac{\Gamma \vdash_+ M : \Pi x : A.\, B \qquad \Gamma \vdash_+ N : A}{\Gamma \vdash_+ M\ N : B[x := N]}\ (\textsc{App}^+) \qquad \frac{\Gamma, x : A \vdash_+ M : B}{\Gamma \vdash_+ \lambda x : A.\, M : \Pi x : A.\, B}\ (\textsc{Lam}^+)$$

$$\frac{\Gamma \vdash_+ A : \square_i^\alpha \qquad \Gamma \vdash_+ B : \square_i^\beta \qquad \mathcal{D}_\to(\gamma) = (\alpha, \beta)}{\Gamma \vdash_+ A \to B : \square_i^\gamma}\ (\textsc{Arrow}^+)$$

$$\frac{\Gamma \vdash_+ A : \square_i^\alpha \qquad \Gamma, x : A \vdash_+ B : \square_i^\beta \qquad \mathcal{D}_\Pi(\gamma) = (\alpha, \beta)}{\Gamma \vdash_+ \Pi x : A.\, B : \square_i^\gamma}\ (\textsc{Pi}^+)$$

Fig. 9. Typing rules for CC_ω^+

$$\begin{aligned} |\square_i^\alpha|^- &:= \square_i & |\langle\rangle|^- &:= \langle\rangle \\ |\Pi x : A.\, B|^- &:= \Pi x : |A|^-.\, |B|^- & |\Gamma, x : A|^- &:= \Gamma, x : |A|^- \\ |\lambda x : A.\, B|^- &:= \lambda x : |A|^-.\, |B|^- \\ |T\ U|^- &:= |T|^-\ |U|^- \\ |x|^- &:= x \end{aligned}$$

Fig. 10. Erasure function from CC_ω^+ to CC_ω

Theorem 6.3 (Conservativity of CC_ω^+).

$$\forall \Gamma \in \mathcal{C}_{CC_\omega^+},\ \forall t, A \in \mathcal{T}_{CC_\omega^+}, \quad \Gamma \vdash_{CC_\omega^+} t : A \implies |\Gamma|^- \vdash_{CC_\omega} |t|^- : |A|^-$$

Proof. By induction on the derivation. □

6.3 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 § 6.2. This analogue is geared towards proof transfer, as discussed in § 3.1, and therefore designed to *synthesize* the output of the translation from its input, rather than to *check* that certain pairs of terms are 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 amount of data in the output of the translation. In the Trocq calculus, this extra information is a type of CC^+_ω.

We thus define Trocq *contexts* $\mathcal{R}_{CC^+_\omega}$ as lists of quadruples:

$$\Delta ::= \langle\rangle \mid \Delta,\ x\ @\ A\ \sim\ x'\ \because\ x_R \quad \text{where } A \in \mathcal{T}_{CC^+_\omega}.$$

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

$$\mathrm{Var}\,(\langle\rangle) = \langle\rangle \qquad \mathrm{Var}(\Delta,\ x\ @\ A\ \sim\ x'\ \because\ x_R) = \mathrm{Var}(\Delta), x, x', x_R.$$

A Trocq context Δ is *well-formed*, written $\Delta \vdash_+$, if the sequence $\mathrm{Var}(\Delta)$ is duplicate-free. In this case, we use the notation $\Delta(x) = (A, x', x_R)$ as a synonym of $(x, A, x', x_R) \in \Xi$.

Now, a Trocq judgment is a 4-ary relation of the form $\Delta \vdash_t t\ @\ A\ \sim\ t'\ \because\ t_R$, which is read *in context Δ, term t of annotated type A translates to term t', because t_R* and t_R is called a parametricity witness. Trocq judgments are defined by the rules of Figure 11. This definition involves a weakening function for parametricity witnesses, defined as follows.

Definition 6.4. For all $p, q \in \{0, 1, 2_a, 2_b, 3, 4\}$, such that $p \geq q$, we define map $\downarrow^p_q : \mathsf{M}_p \to \mathsf{M}_q$, 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 $\Downarrow^\alpha_\beta : \boxdot^\alpha\ A\ B \to \boxdot^\beta\ A\ B$ is defined by:

$$\Downarrow^{(m,n)}_{(p,q)}\ \langle R, M^{\rightarrow}, M^{\leftarrow}\rangle := \langle R, \downarrow^m_p M^{\rightarrow}, \downarrow^n_q M^{\leftarrow}\rangle.$$

The weakening function on parametricity witnesses is defined on Figure 12 by extending function $\Downarrow^\alpha_\beta$ to all relevant pairs of types of CC^+_ω , *i.e.*, $\Downarrow^T_U$ is defined for $T, U \in \mathcal{T}_{CC^+_\omega}$ as soon as $T \preccurlyeq U$.

We first show the following standardization lemma for Trocq judgments.

Lemma 6.5. *If $\Delta \vdash_t u\ @\ A\ \sim\ u'\ \because\ u_R$ holds, then there exists a derivation of this judgment where the* TrocqConv *rule is used only on variables, i.e., with t, t' and t_R variables.*

Proof. By induction on the derivation. The only interesting case is the TrocqConv rule. By induction hypothesis the premise can be turned into a derivation D which ends either with TrocqConv rule on a variable or ends with any rule but the TrocqConv rule. There are then four nontrivial cases for this last rule of D:

- TrocqConv rule on a variable: the two successive TrocqConv rules collapse into a single one by composing weakenings;
- TrocqSort rule: the TrocqConv rule can be omited as, by construction, $\Downarrow^\beta_\delta p^{\alpha,\beta}_\square = p^{\alpha,\delta}_\square$;
- TrocqPi rule: similar, since by construction $\Downarrow^\beta_\delta p^\beta_\Pi = p^\delta_\Pi$;
- TrocqArrow rule: similar, since by construction $\Downarrow^\beta_\delta p^\beta_\rightarrow = p^\delta_\rightarrow$.

□

Lemma 6.6. *The relation associating a pair* $(t, A) \in {CC^+_\omega}^2$ *with a pair* $(t', t_R) \in {CC^+_\omega}^2$ *such that* $\Delta \vdash_t t \mathbin{@} A \sim t' \mathrel{\because} t_R$ *holds, with* Δ *a well-formed* Trocq *context, is* functional. *More precisely, for any well-formed* Trocq *context* Δ*:*

$$\forall \Delta, t, A, t', u', t_R, u_R, \quad \Delta \vdash t \mathbin{@} A \sim t' \mathrel{\because} t_R \quad \wedge \quad \Delta \vdash t \mathbin{@} A \sim u' \mathrel{\because} u_R \implies (t', t_R) = (u', u_R)$$

Proof. First we use Lemma 6.5 to normalise the derivations. Then we reason by induction on the syntax of t. Because the derivation is normal, there is only one possible rule for each syntactic construction, except for the TrocqArrow and TrocqPi rules, but p^δ_Π and $p^\delta_\to$ happen to coincide when they are both applicable. □

We introduce a conversion function γ from Trocq contexts to CC^+_ω contexts:

$$\begin{aligned} \gamma(\langle\rangle) &= \langle\rangle \\ \gamma(\Delta,\ x \mathbin{@} A \sim x' \mathrel{\because} x_R) &= \gamma(\Delta),\ x : A, x' : A', x_R : \mathsf{rel}(A_R)\ x\ x' \\ &\quad \text{where} \quad \Delta \vdash_t A \mathbin{@} \square^\alpha_i \sim A' \mathrel{\because} A_R. \end{aligned}$$

Note that γ is well defined because of Lemma 6.6.

$$\frac{(\alpha, \beta) \in \mathcal{D}_\square}{\Delta \vdash_t \square^\alpha_i \mathbin{@} \square^\beta_{i+1} \sim \square^\alpha_i \mathrel{\because} p^{\alpha,\beta}_{\square_i}} \text{(TrocqSort)} \qquad \frac{(x, A, x', x_R) \in \Delta \qquad \gamma(\Delta) \vdash_+}{\Delta \vdash_t x \mathbin{@} A \sim x' \mathrel{\because} x_R} \text{(TrocqVar)}$$

$$\frac{\Delta \vdash_t t \mathbin{@} \Pi x : A.\, B \sim t' \mathrel{\because} t_R \qquad \Delta \vdash_t u \mathbin{@} A \sim u' \mathrel{\because} u_R}{\Delta \vdash_t t\ u \mathbin{@} B[x := u] \sim t'\ u' \mathrel{\because} t_R\ u\ u'\ u_R} \text{(TrocqApp)}$$

$$\frac{\begin{array}{c}\Delta \vdash_t A \mathbin{@} \square^\alpha_i \sim A' \mathrel{\because} A_R \\ \Delta, x \mathbin{@} A \sim x' \mathrel{\because} x_R \vdash_t t \mathbin{@} B \sim t' \mathrel{\because} t_R\end{array}}{\Delta \vdash_t \lambda x : A.\, t \mathbin{@} \Pi x : A.\, B \sim \lambda x' : A'.\, t' \mathrel{\because} \lambda x\, x'\, x_R.\, t_R} \text{(TrocqLam)}$$

$$\frac{\begin{array}{c}(\alpha, \beta) = \mathcal{D}_\to(\delta) \\ \Delta \vdash_t A \mathbin{@} \square^\alpha_i \sim A' \mathrel{\because} A_R \qquad \Delta \vdash_t B \mathbin{@} \square^\beta_i \sim B' \mathrel{\because} B_R\end{array}}{\Delta \vdash_t A \to B \mathbin{@} \square^\delta_i \sim A' \to B' \mathrel{\because} p^\delta_\to\ A_R\ B_R} \text{(TrocqArrow)}$$

$$\frac{\begin{array}{c}(\alpha, \beta) = \mathcal{D}_\Pi(\delta) \qquad \Delta \vdash_t A \mathbin{@} \square^\alpha_i \sim A' \mathrel{\because} A_R \\ \Delta, x \mathbin{@} A \sim x' \mathrel{\because} x_R \vdash_t B \mathbin{@} \square^\beta_i \sim B' \mathrel{\because} B_R\end{array}}{\Delta \vdash_t \Pi x : A.\, B \mathbin{@} \square^\delta_i \sim \Pi x' : A'.\, B' \mathrel{\because} p^\delta_\Pi\ A_R\ B_R} \text{(TrocqPi)}$$

$$\frac{\Delta \vdash_t t \mathbin{@} A \sim t' \mathrel{\because} t_R \qquad \gamma(\Delta) \vdash_+ A \lesssim B}{\Delta \vdash_t t \mathbin{@} B \sim t' \mathrel{\because} \Downarrow^A_B t_R} \text{(TrocqConv)}$$

Fig. 11. Trocq rules

An abstraction theorem relates Trocq judgments and typing in CC^+_ω.

$$\Big\Downarrow^{\Box_i^\alpha}_{\Box_i^{\alpha'}} t_R := \Downarrow^{\alpha}_{\alpha'} t_R \qquad \Big\Downarrow^{A\ u}_{A'\ u'} t_R := \Big\Downarrow^{A}_{A'} u\ u'\ t_R \qquad \Big\Downarrow^{\lambda x{:}A.\,B}_{\lambda x{:}A'.\,B'} u\ u'\ t_R := \Big\Downarrow^{B[x:=u]}_{B'[x:=u']} t_R$$

$$\Big\Downarrow^{\Pi x{:}A.\,B}_{\Pi x{:}A'.\,B'} t_R := \lambda x\ x'\ x_R.\ \Big\Downarrow^{B}_{B'} \left(t_R\ x\ x'\ \left(\Big\Downarrow^{A'}_{A} x_R\right)\right) \qquad \Big\Downarrow^{A}_{A'} t_R := t_R$$

Fig. 12. Weakening of parametricity witnesses

Theorem 6.7 (Trocq abstraction theorem).

$$\frac{\gamma(\Delta) \vdash_+ \qquad \gamma(\Delta) \vdash_+ t : A \qquad \Delta \vdash_t t\ @\ A \sim t' \mathrel{\because} t_R \qquad \Delta \vdash_t A\ @\ \Box_i^\alpha \sim A' \mathrel{\because} A_R}{\gamma(\Delta) \vdash_+ t' : A' \qquad \textit{and} \qquad \gamma(\Delta) \vdash_+ t_R : \mathit{rel}(A_R)\ t\ t'}$$

Proof. By induction on derivation $\Delta \vdash_t t\ @\ A \sim t' \mathrel{\because} t_R$. □

Note that type A in the typing hypothesis $\gamma(\Delta) \vdash_+ t : A$ 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 6.8. Since by definition of $p_\Box^{\alpha,\beta}$ (Equation 18), we have $\vdash_t \Box^\alpha\ @\ \Box^\beta \sim \Box^\alpha \mathrel{\because} p_\Box^{\alpha,\beta}$, by applying Theorem 6.7 with $\gamma(\Delta) \vdash_+ A : \Box^\alpha$, we get:

$$\frac{\gamma(\Delta) \vdash_+ A : \Box^\alpha \qquad \Delta \vdash_t A\ @\ \Box^\alpha \sim A' \mathrel{\because} A_R}{\gamma(\Delta) \vdash_+ A_R : \mathsf{rel}(p_\Box^{\alpha,\beta})\ A\ A'}.$$

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

Remark 6.9. By applying Remark 6.8 with $\vdash_+ \Box^\alpha : \Box^\beta$, we indeed obtain that $\vdash_+ p_\Box^{\alpha,\beta} : \boxdot^\beta\ \Box^\alpha\ \Box^\alpha$ as expected, provided that $(\alpha, \beta) \in \mathcal{D}_\Box$.

6.4 Conservativity over raw and univalent parametricity

We show that Trocq entails raw parametricity and univalent parametricity by defining minimal and maximal annotations $|\cdot|^\top : \mathcal{T}_{CC_\omega} \to \mathcal{T}_{CC^+_\omega}$ and $|\cdot|^\bot : \mathcal{T}_{CC_\omega} \to \mathcal{T}_{CC^+_\omega}$ as in Figure 13, which are two obvious right inverses of $|\cdot|^-$. We then have the following theorems.

Theorem 6.10 (Trocq conservativity over raw parametricity).

$$\begin{aligned}\forall \Gamma \in \mathcal{C}_{CC_\omega}, \forall \Xi \in \mathcal{R}_{CC_\omega}, \forall t, A, t', t_R \in \mathcal{T}_{CC_\omega}, \quad & \Gamma \triangleright \Xi \ \wedge\ \Gamma \vdash t : A \ \wedge\ \Xi \vdash t \sim t' \mathrel{\because} t_R \\ \Longrightarrow \quad & \exists t_\bot, \quad |\Gamma; \Xi|^\bot \vdash_t |t|^\bot\ @\ |A|^\bot \sim \left|t'\right|^\bot \mathrel{\because} t_\bot\end{aligned}$$

and

$$\begin{aligned}\forall \Gamma \in \mathcal{C}_{CC_\omega}, \forall \Xi \in \mathcal{R}_{CC_\omega}, \forall t, A, t', t_\bot \in \mathcal{T}_{CC_\omega}, \quad & \Gamma \triangleright \Xi \ \wedge \\ & |\Gamma; \Xi|^\bot \vdash_t |t|^\bot\ @\ |A|^\bot \sim \left|t'\right|^\bot \mathrel{\because} t_\bot \\ \Longrightarrow \quad & \exists t_R, \quad \Gamma \vdash t : A \ \wedge\ \Xi \vdash t \sim t' \mathrel{\because} t_R\end{aligned}$$

Proof. By induction on the derivation. □

Theorem 6.11 (Trocq conservativity over univalent parametricity).

$$\forall \Gamma \in \mathcal{C}_{CC_\omega}, \forall \Xi \in \mathcal{R}_{CC_\omega}, \forall t, A, t', t_u \in \mathcal{T}_{CC_\omega}, \quad \Gamma \triangleright \Xi \ \wedge\ \Gamma \vdash t : A \ \wedge\ \Xi \vdash_u t \sim t' \ \because\ t_u$$
$$\implies \quad \exists t_\top, \quad |\Gamma; \Xi|^\top \vdash_t |t|^\top \ @\ |A|^\top \sim \left|t'\right|^\top \ \because\ t_\top$$

and

$$\forall \Gamma \in \mathcal{C}_{CC_\omega}, \forall \Xi \in \mathcal{R}_{CC_\omega}, \forall t, A, t', t_\top \in \mathcal{T}_{CC_\omega}, \quad \Gamma \triangleright \Xi \wedge$$
$$|\Gamma; \Xi|^\top \vdash_t |t|^\top \ @\ |A|^\top \sim \left|t'\right|^\top \ \because\ t_\top$$
$$\implies \quad \exists t_u, \quad \Gamma \vdash t : A \ \wedge\ \Xi \vdash_u t \sim t' \ \because\ t_u$$

Proof. By induction on the derivation. □

For the sake of simplicity we do not formally state the relation between witnesses t_R and $t_\perp$ on one hand, and witnesses t_u and $t_\top$ on the other. But they pairwise have the same underlying relation when it makes sense. This is a consequence of Remark 5.9.

For $\alpha \in \{\top, \perp\}$, we define

$$|\Box_i|^\alpha := \Box_i^\alpha$$
$$|\Pi x : A.\, B|^\alpha := \Pi x : |A|^\alpha .\, |B|^\alpha$$
$$|\lambda x : A.\, B|^\alpha := \lambda x : |A|^\alpha .\, |B|^\alpha$$
$$|M\ N|^\alpha := |M|^\alpha\ |N|^\alpha$$
$$|x|^\alpha := x$$

$$|\langle\rangle|^\alpha := \langle\rangle$$
$$\left|\Gamma, x : A, x' : A', x_R : A_R\ x\ x'\right|^\alpha := |\Gamma|^\alpha, x : |A|^\alpha$$

$$|\langle\rangle; \langle\rangle|^\alpha := \langle\rangle$$
$$\left|(\Gamma, x : A, x' : A', x_R : A_R\ x\ x')\ ;\ (\Xi, x \sim x' \ \because\ x_R)\right|^\alpha := |\Gamma; \Xi|^\alpha,\ x\ @\ |A|^\alpha \sim x' \ \because\ x_R$$

Fig. 13. Annotation function from CC_ω to CC_ω^+

6.5 Constants and constraints

As was the case for the raw and univalent parametricity translations, concrete applications require extending Trocq with constants. In this case as well, we treat constants in a similar way as variables, except that the latter are stored in a global context instead of a typing context. A crucial difference though is that a same constant may lead to several copies in this global CC_ω^+ context, with different annotated types.

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, `list` can be annotated with type $\Box^\alpha \to \Box^\alpha$ for any $\alpha \in \mathcal{A}$.

We thus need to extend the typing of constants to allow for *sets* of annotated types. Hence, given

- a set $\mathcal{C}$ of symbols,
- for each $c \in \mathcal{C}$, a term $T_c \in \mathcal{T}_{CC_\omega}$, such that $\vdash T_c : \square$,
- for each $c \in \mathcal{C}$, a subset $T_c^+ \subset \mathcal{T}_{CC_\omega^+}$ of annotated types, such that for all c and for all $A \in T_c^+$, $\vdash |A|^- \equiv T_c$,
- a partial translation function $\mathcal{D}_. : \mathcal{C} \to \mathcal{T}_{CC_\omega^+} \to \mathcal{C}^2$ such that if for all $c, c', c_R \in \mathcal{C}$ and $A \in CC_\omega^+$ such that $\mathcal{D}_c(A) = (c', c_R), \vdash T_c \ @ \ \square^\alpha \sim T_{c'} \because T_{c_R}$ holds for some α and $\vdash c_R : \mathsf{rel}(T_{c_R})\ c\ c'$ holds,

the rules of Figure 14 shall respectively extend CC_ω^+ and Trocq.

$$\frac{c \in \mathcal{C} \qquad A \in T_c^+}{\Gamma \vdash c : A}\ (\textsc{Const}^+) \qquad\qquad \frac{\mathcal{D}_c(A) = (c', c_R)}{\Delta \vdash_t c \ @ \ A \sim c' \because c_R}\ (\textsc{TrocqConst})$$

Fig. 14. Additional constant rules for CC_ω^+ and Trocq

Every constant declared in the global environment thus 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.*, $\forall c, \forall A, \forall B,\ A, B \in T_c^+ \Rightarrow |A|^- = |B|^-$, by making sure their erasure coincides with the CC_ωtype of the constant c. For example, $T_{\texttt{list}}^+ = \{\square^\alpha \to \square^\alpha \mid \alpha \in \mathcal{A}\}$.

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, the translation $\mathcal{D}_{\texttt{list}}(\square^{(1,0)} \to \square^{(1,0)})$ of the `list` inductive type is equal to $(\texttt{list},\ \lambda A\, A'\, A_R.\ (\texttt{List.All2}\ A_R,\ \texttt{List.map}\ (\mathsf{map}\ A_R)))$, 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 $\mathsf{map}\ A_R : A \to A'$ extracts the *map* projection of the tuple A_R of type $\boxdot^{(1,0)}\ A\ A' \equiv \Sigma R.\, A \to A'$. Part of these translations can be generated automatically by weakening.

Note that for an input term featuring constants, an unfortunate choice of annotation in the derivation tree may lead to a stuck translation. For example, we cannot pick the annotated type $\square^{(1,0)} \to \square^{(1,0)}$ in order to translate $\texttt{list}\ A \ @ \ \square^{(2_a,0)}$. We describe in Section 7 the algorithm which picks correct levels for each constant annotation.

7 Implementation

The Trocq plugin [22] turns the material presented in Section 6 into an actual tactic, called `trocq`, for automating proof transfer in Rocq/Coq. This tactic synthesizes a new goal from the current one, as well as a proof that the former implies the latter. Otherwise said, the tactic operates a translation at level $(1, 0)$ of the current goal. 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 [29, 54], on top of Rocq/Coq libraries formalizing the contents of Section 5. In the logic programming paradigm of Elpi, each rule of Figure 11 translates gracefully into a corresponding λProlog predicate, making the corresponding source code very close to the presentation of §6.3. However, the Trocq plugin also implements a much less straightforward annotation inference algorithm, so as to hide the management of sort annotations to Rocq/Coq users completely.

The algorithm proceeds in three phases. First, it annotates terms, with unknown annotations left as meta-variables (§7.1). Second, it runs the direct translation of the Trocq calculus as a logic program, leaving meta-variables for every term which relies on unknown annotations, while storing constraints on annotations in a graph (§7.2). Last, it solves the constraint graph by assigning the minimal possible value for each annotation, and fills the remaining holes accordingly (§7.3). In addition, the plugin is relying on user given translations for constants (§7.4).

Two variants of the plugin are currently available, one based on the HoTT library[3] [57], and the other based on Rocq/Coq's standard library[4]. Since some translations are only possible in the presence of the axiom of univalence or functional extensionality, we provide two commands, namely `Trocq Register Univalence u` and `Trocq Register Funext f` for users wanting to rely on these axioms. Naturally, the standard libarary version of Rocq/Coq is incompatible with the use of the former.

7.1 Encoding CC^+_ω .

To implement the annotation calculus CC^+_ω , we simply annotate Rocq/Coq's sort `Type` with a pair (n, m) using *convertible* synonyms `PType n m` , where `PType := fun (_ _ : map_class) => Type` . The two thrown-away arguments code for the parametricity class annotations using the `map_class` inductive type with values ranging from `map0` to `map4`, representing the values in $\mathcal{A}$. They are phantom values whose single purpose is that we can match them syntactically in the plugin. Since `PType m n` is definitionally equal to `Type` the annotation can be erased in a step of δ-reduction and annotated terms are valid Rocq/Coq terms, which means Rocq/Coq typechecker will be unaffected.

The plugin first translates all occurrences of `Type` into `PType _ _` , and during the run of the second part of the algorithm, the two holes of each `PType` will be related to other levels in the constraint graph, according to the rules given by $\mathcal{D}_\square$, $\mathcal{D}_\Pi$, and $\mathcal{D}_c$ for any constant c.

7.2 Synthesis

The logic programming paradigm on which Elpi is based, is ideal to implement algorithms expressed as inference rules, as each rule can be associated to an instance of a predicate. The linear traversal of the input term at the core of the Trocq plugin is operated by the predicate `param` , of arity 4, where `param X T X' XR` stands for the parametricity sequent $\Delta \vdash_t x \ @ \ T \sim x' \ \because \ x_R$ for a certain context Δ. In this sequent, x and T are input values (initially, the source goal and the annotated sort $\square^{(0,1)}$), and the synthesized term x' and witness x_R are outputs. Each construct of CC_ω leads to *one* instance of the predicate. As an example, let us inspect the instance of the `param` predicate for dependent products, which implements the rule TrocqPi of the Trocq calculus. For the sake of readability, we removed lines related to logs, pretty-printing, and fresh universe instance generation. The head of the predicate is:

```
param (prod N A B) (app [pglobal (const PType) _, M1, M2]) Prod' ProdR :-
  param.db.ptype PType, !,
  cstr.univ-link C M1 M2,
```

which matches an input term $\Pi x : A. B$ and our Rocq/Coq encoding of its annotated type $\square^{(M_1,M_2)}$. Then, following the hypotheses in the inference rule, the predicate computes the prescribed annotation $(C_A, C_B) = \mathcal{D}_\Pi(C)$, and does two recursive calls on A and B with classes C_A and C_B:

```
cstr.dep-pi C CA CB,
cstr.univ-link CA M1A M2A,
param A (app [pglobal (const PType) _, M1A, M2A]) A' AR,
cstr.univ-link CB M1B M2B,
TB = app [pglobal (const PType) _, M1B, M2B],
@annot-pi-decl N A a\ pi a' aR\ param.store a A a' aR =>
  param (B a) TB (B' a') (BR a a' aR),
```

[3]https://github.com/coq-community/trocq/releases/tag/0.1.6
[4]https://github.com/coq-community/trocq/releases/tag/0.1.8%2Bprop

The HOAS encoding of Rocq/Coq terms saves us from an otherwise tricky management of bound variables.

The last step (omitted in the above snippet) is to build the output proof $p^C_\Pi\ A_R\ B_R$. As the axioms (univalence, functional extensionality) that might be involved in some proofs are not assumed globally, they are used as an additional argument albeit only in the building blocks that require them. Therefore, we check whether the requested rule requires the addition of an axiom to the list of arguments (in the case of the dependent product, function extensionality). If this axiom is not present in the context at the time of calling this part of the code, the tactic rightfully fails, because the translation is impossible.

Exploiting symmetries. Trocq provides several distinct rules per language construct (such as Π) and per relation structure among the 36 items in the hierarchy: for a same construct, these rules differ by the annotations required on the input of the rule, and by the structure of the relation relating the input term and the synthesized one. For each such rule, a Rocq/Coq function provides the corresponding rule building block. Making the most of symmetries, the 495 rule building blocks are generated by meta-programming from only 9 manually defined ones.

7.3 Solving constraints

Finally, the traversal of the input term collects *constraints* on the annotations, as multiple valid solutions might exist: for instance, an implication might be obtained from weakening an equivalence.

The constraints are stored as a graph where nodes are either variable or constant parametricity classes, and different types of edges exist, based on the constraints involved in the translation. An example of constraint graph is available in Figure 15. For instance, the edges from X_1 to X_2 and X_3 represent the constraint $\mathcal{D}_\Pi(X_1) = (X_2, X_3)$, and the edge from X_6 to the constant class $(3, 2_\mathrm{b})$ represents the constraint $(3, 2_\mathrm{b}) \geq X_6$.

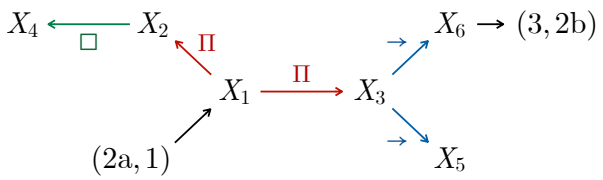


Fig. 15. Example of constraint graph

The algorithm strives to minimize the requirements on the user-defined building blocks, which also amounts to minimizing the dependency on axioms.

This inference procedure is formalized as a *finite domain* constraint solving problem, and implemented using *Constraint Handling Rules* (CHR) language [30], as available in Elpi. It allows suspending goals and turning them into syntactic constraints, to be reduced with meta-level rules or resumed when some variables are instantiated. The CHR language is well suited to prototype constraint solvers because the core ingredient of such solvers, constraint propagation and consistency checking, can be implemented as CHR reduction rules.

7.4 Handling of constants

In order to handle constants, we provide a vernacular command `Trocq Use` to insert new Trocq quadruples in the database of known rules. Given a term t_R of type $R\ t\ t'$, we insert an annotated type A from R and insert the rule $t\ @\ A \sim t' \because t_R$ in the database.

We provide an API to create new relations out of section retraction pairs, reflecting Remark 5.10.

```
Record SplitSurj.type (A B : Type) : Type := Build {
  retract :> A → B;
  section : B → A;
  sectionK : ∀ x : B, retract (section x) = x
}.

Record SplitInj.type (A B : Type) : Type := Build {
  section :> A → B;
  retract : B → A;
  sectionK : ∀ x : A, retract (section x) = x
}.

Definition SplitSurj.toParam {A B : Type} : SplitSurj.type A B → Param42a.Rel A B := ...

Definition SplitInj.toParam {A B : Type} : SplitInj.type A B → Param42b.Rel A B := ...
```

Here, `Param42a.Rel` is the Rocq/Coq definition for $\boxdot^{(4,2_a)}$, and `Param42b.Rel` for $\boxdot^{(4,2_b)}$.

We also provide various helper lemmas, including lemmas that exploit the symmetry of relations to derive more structure on them.

8 Examples

This section illustrates the usage of the `trocq` tactic on various concrete examples. They rely on the manual Trocq translation of the following inductive types and their constructors and sometimes eliminators, seen as constants: `Empty`, `bool`, `nat`, `option`, `prod`, `sum`, `sigma`, `list`, `eq` / `paths` and `vector`.

All the examples from this section can be dealt with Trocq without using axioms, neither univalence nor functional extensionality.

8.1 Isomorphisms

Bitvectors (code). Here are two possible encodings of bitvectors in Rocq/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 2.2 is to obtain the following elimination scheme, from that available on type $\mathbb{N}$:

```
N_ind : ∀ P : N → □, P O_N → (∀ n : N, P n → P (S_N n)) → ∀ n : N, P n
```

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

```
Definition N_R : Param44.Rel N ℕ := ...
Lemma O_R : rel N_R O_N O_ℕ.
Lemma S_R : ∀ m n, rel N_R m n → rel N_R (S_N m) (S_ℕ n).
Trocq Use N_R O_R S_R.
```

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

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

Inspecting this proof confirms that only information up to level $(2_{\mathrm{a}}, 3)$ 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 $\mathbb{N} \to I$.

8.2 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 $\mathbb{Z}/p\mathbb{Z}$, as pointed out by Example 2.4. Let us show how Trocq can substitute $\mathbb{Z}$ for $\mathbb{Z}_9$ in the following statement:

```
Lemma flt3_step : ∀ m n p : ℤ, (m * n * p % 3 ≠ 0 → m^3 + n^3 ≠ p^3)%ℤ.
```

where scope `%ℤ` is for the usual product, sum, modulo 3 and zero on type `ℤ`, and `≡` is an equality test modulo 9 on type `ℤ`. A call to the `trocq` tactic turns the main goal of the proof into

```
∀ m n p : ℤ₉, (m * n * p % 3 ≠ 0 → m^3 + n^3 ≠ p^3)%ℤ₉.
```

where scope `%ℤ₉` is for the usual product, sum, modulo 3 and zero on type `ℤ₉`. The `trocq` tactic automatically generated the following implication, which deduces a proof on $\mathbb{Z}$ *from* its modular analogues, reducing the quantification on all integers to a bounded quantification:

```
(∀ m n p : ℤ₉, m * n * p % 3 ≠ 0 → m^3 + n^3 ≠ p^3)%ℤ₉ →
  (∀ m n p : ℤ, m * n * p % 3 ≠ 0 → m^3 + n^3 ≠ p^3)%ℤ.
```

Types `ℤ₉` and `ℤ` are obviously not equivalent, but a *retraction* is actually enough for this proof transfer. We have:

```
modp : ℤ → ℤ₉
reprp : ℤ₉ → ℤ
reprpK : ∀ (x : ℤ₉), modp (reprp x) = x
Rp : Param42a.Rel ℤ ℤ₉
```

where `Rp`, (a proof that $\boxdot^{(4,2_a)}$ `ℤ ℤ₉`), is obtained from `reprpK` via lemma `SplitSurj.toParam`. Performing this first step of the proof of lemma `flt3_step` by `trocq` now just requires relating the respective zeroes, additions, multiplications, reduction modulo 3 and equalities of the two types:

```
R0 : Rp 0%ℤ 0%ℤ₉.
Radd : ∀ (m : ℤ) (x : ℤ₉) (xR : Rp m x) (n : ℤ) (y : ℤ₉) (yR : Rp n y), Rp (m + n)%ℤ (x + y)%ℤ₉.
Rmul : ∀ (m : ℤ) (x : ℤ₉) (xR : Rp m x) (n : ℤ) (y : ℤ₉) (yR : Rp n y), Rp (m * n)%ℤ (x * y)%ℤ₉.
Rmod3 : ∀ (n : ℤ) (x : ℤ₉) (xR : Rp n x),Rp (n % 3)%ℤ (x % 3)%ℤ₉.
Reqmodp : ∀ (m : ℤ) (x : ℤ₉) (xR : Rp m x) (n : ℤ) (y : ℤ₉) (yR : Rp n y),
           Param01.Rel (m ≡ n) (x = y).
Trocq Use Rp R0 Radd Rmul Rmod3 Reqmodp. (* informing Trocq with these relations *)
```

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

```
  ∀ (m n : ℤ), modp (m * n)%ℤ = (modp m * modp n)%ℤ₉.
```

Summable sequences. (code). Given a sequence $(u_n)_{n\in\mathbb{N}}$ of non-negative real numbers, *i.e.*, a function $u : \mathbb{N} \to [0, +\infty[$, 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:

$$\sum(u+v) = \sum u + \sum v \tag{19}$$

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 $u : \mathbb{N} \to [0, +\infty]$, 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 19 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 [43], Isabelle/HOL's Archive of Formal Proofs, coq-interval [42] or Rocq/Coq's mathcomp-analysis [1], resort to such extended domains and transfer steps, notably for defining measure theory. Yet, as reported by expert users [33], the associated transfer bureaucracy is essentially done manually and thus significantly clutters formal developments in real and complex analysis, probabilities, etc. We show how to deal with this example with Trocq.

This example 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 R̄≥0 : Type := Fin : ℝ≥0 → R̄≥0 | Inf : R̄≥0.
Definition seqℝ≥0 := nat → ℝ≥0. Definition seqR̄≥0 := nat → R̄≥0.
Record summable := {to_seq :> seqℝ≥0; _ : isSummable to_seq}.
```

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

```
Definition Rrseq (u : summable) (v : seqR̄≥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 $\mathtt{\Sigma_{\overline{\mathbb{R}}_{\geq 0}}\ u : \overline{\mathbb{R}}_{\geq 0}}$ is the sum of a sequence $\mathtt{u : seq_{\overline{\mathbb{R}}_{\geq 0}}}$ 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 Σℝ≥0 (u : summable) : ℝ≥0 := truncate (Σ_R̄≥0 (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 ℝ≥0 and R̄≥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 : ∀ (u v : R̄≥0), Σ_R̄≥0 (u + v) = Σ_R̄≥0 u + Σ_R̄≥0 v.
Proof.(...) Qed. (* easy, as no convergence proof is needed *)

Lemma sum_nnR_add : ∀ (u v : ℝ≥0), Σℝ≥0 (u + v) = Σℝ≥0 u + Σℝ≥0 v.
Proof. trocq; exact sum_xnnR_add. Qed.
```

8.3 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 : ∀ A A' (AR : Param44.Rel A A'), Param44.Rel (list A) (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 `ℤ` and `ℤ₉` from Section 8.2, 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 ℤ₉` to `list ℤ`.

```
listR2a4 : ∀ A A' (AR : Param2a4.Rel A A'),
  Param2a4.Rel (list A) (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 Rocq/Coq's standard library, as follows.

```
Definition tuple (A : Type) : nat → Type := fix F n :=
  match n with O => 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 : ∀ A n, tuple A (S n) → A`, and a function `const : ∀ A, A → ∀ n, tuple A n` creating a constant vector, and simultaneously refines integers into the integers modulo p from Section 8.1:

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

Lemma head_cst' : ∀ (n : nat) (z : ℤ₉), head (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 Rocq/Coq's standard library. Note that the proof uses the equivalence between `Vector.t` and `tuple` but only requires a retraction between parameter types.

9 Related work

The literature dealing with proof transfer in interactive theorem proving roughly falls in two camps. A first line of work concerns the design of practical tools for automating the synthesis of mundane parts of formal developments by *reusing* some pre-existing formal proofs to inform this synthesis. The tools may operate on proof scripts or on proof terms, the latter being more directly connected to the present work.

In this context, the synthesis procedure is most often heuristical. In particular, it is not expected to produce any formal guarantee of the relation between the pre-existing proof term and the synthesized one, as type-checking suffices to validate a candidate proof term. Barthe and Pons [11] already noticed that the computational content of type isomorphisms can serve proof transfer. The first implementation report of a tool based on this idea appeared soon after [41]. This meta-program implemented a proof rewriting heuristic producing, from an existing proof term of a certain statement, a candidate proof term of a modified statement, with no formal guarantee – not even that of being well-typed. The rise of interactive theorem proving for program verification,

in particular outside academia, highlighted the crucial need for better formal proof engineering methods and tools. In particular, *proof repair* techniques aim at providing a better support for synchronizing an evolving code base with its formal guarantees of correctness. This question is taken in a broader sense than the formalization of proof transfer provided in Section 3.1, as it may concern programs written outside of the logic, and may involve inferring program alignments, invariants, etc. [32]. Some of the corresponding tools however are designed to repair proofs of internal programs to the type theory backing the interactive prover. For instance, Pumpkin Pi provides a comprehensive proof repair toolset for *repairing* proofs after changing certain datatypes for equivalent ones [49, 60].

A second line of work corresponds to proof transfer in the sense of Section 3.1: by contrast with the previous angle, a central issue in this case is the synthesis of a proof term witnessing that two certain terms are related. Generalized rewriting [50], which generalizes setoid rewriting to preorders, is actually a variant of proof transfer, albeit within the same type. As such, it allows in particular rewriting under binders. The restriction to homogeneous relations however excludes more general instances of proof transfer. For instance, datatype representation change and quasi-PERs (QPER, or zig-zag complete relations) [37] are inherently heterogeneous.

The other proof transfer methods we are aware of all address the case of heterogeneous relations. Incidentally, they can thus also be used for the homogeneous case, and therefore for generalized rewriting, although this special case is seldom emphasized. The Coq Effective Algebra Library (CoqEAL) [23, 28] and the Isabelle/HOL transfer package [34, 35, 38, 39], pioneered the use of parametricity-based methods for proof transfer, motivated by the refinement of proof-oriented data-structures to computation-oriented counterparts. Together with a subsequent generalization of the CoqEAL approach [64], these tools address the case of a transfer between a subtype of a certain type A and a quotient of a certain type B, *i.e.*, the case of a trivial QPER in which the zig-zag morphism is a surjection from A to B.

Modern approaches to proof transfer rely on univalence, either as an axiom, in the case of univalent parametricity [53] or as a computing primitive [8]. Key ingredients of univalent parametricity were already present in earlier seemingly unpublished work [6], implemented using an ancestor of the MetaCoq library [51]. Another recent line of work consists in studying to which extent the structural propagation of certain relations can be internalized in a type theory [4, 40, 58].

The columns of Table 1 list the aforementioned tools in chronological order, and they indicate when the features listed as lines are available (✓), not available (✗) or only partially available (🖉). By partially available we mean that the given tool may handle subcases of the given feature (for example Trocq currently only handles subrelations in our hierarchy). The first line distinguishes methods than can address *heterogeneous relations* from those restricted to homogeneous ones. While the oldest tool operates via a monolithic translation of an input proof term, others rather prove an *internal* implication lemma. *Anticipation* [53] refers to the need to define a dedicated structure for the signature to be transported. *Binders* (∀) can prevent transfer, as well as *dependent types*. The latter generally requires a univalence principle, which can be a global assumption or, as in the case of Trocq, an optional ingredient.

10 Conclusion

The Trocq framework can be seen as a generalization of both the raw [15, 36] and the univalent [53] parametricity translations. In particular, it allows for combining and for synthesizing proofs of a whole palette of relations between arbitrary types, when the univalent translation can only deal with equivalences. This framework implements a fine-grained analysis of the resources needed to compute an expected translation. Incidentally, the framework actually provides a better variant of univalent parametricity, since the latter is able to detect when the univalence axiom is not needed,

	Magaud [41]	Setoid rewrite [50]	CoqEAL [23]	Transfer [38]	Zimmermann et al. [64]	UPARAM [53]	Int. Repr. Indep. [8]	Trackt [17]	TROCQ
Heterogen. rel.	✓	✗	✓	✓	✓	✓	✓	✓	✓
Internal	✗	✓	✓	✓	✓	✓	✓	✓	✓
No anticipation	✓	✓	✓	✓	✓	✓	✗	✓	✓
Under ∀	✓	✓	✗	✓	✓	✓	✓	✓	✓
Dep. types	✓	✗	✗	✗	✗	✓	✓	✗	✓
Univalence-free	✓	✓	✓	✓	✓	✗	✗	✓	✓
Subrelations	✗	✓	✗	✗	✗	✗	✗	✗	🩹
QERs	✗	🩹	🩹	🩹	🩹	✗	🩹	✗	🩹
Subtyping	✗	✗	🩹	🩹	🩹	✗	✗	🩹	🩹
	Coq	Coq	Coq	Isabelle/ HOL	Coq	HoTT	Cubical Agda	Rocq / Coq	Rocq/Coq or HoTT

Table 1. Comparison of proof transfer automation devices

when the original variant pulls it in by default. This scrutiny allows in particular to get rid of the univalence axiom for a larger class of equivalence proofs [52], and to deal with refinement relations for arbitrary terms, unlike the CoqEAL library [23].

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. Altenkirch and Kaposi already proposed a symmetrical, skolemized phrasing of type equivalence [5], but for different purposes. Definition 5.4 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. In practice, an important feature of the resulting hierarchy of structures for relations is that weaker structures are literally *included* in richer ones. This feature is indeed a crucial invariant for hierarchies implemented in intensional dependent type theory, which should preserve the definitional confluence of inheritance paths [2]. As such, it fulfills the standards enforced by principled instruments for the management of hierarchies [24], although current, independent technical limitations of the latter regarding universe polymorphism prevented us from using them in our plugin. We conjecture that this hierarchy can be enriched further with more interesting structures for relations.

Another obvious future direction of improvement for TROCQ is to extend it to the full calculus of constructions with inductive types (CIC). This requires providing an automated way to produce families of TROCQ translations for inductive types, indexed by the desired output annotations. Raw parametricity translations of for CIC have been described for several variants of inductive type presentations [13, 14, 36], but not their extension to univalent parametricity. The gallery of examples available with the TROCQ plugin includes a manual translation for several inductive types with parameters and indices such as `Empty`, `bool` (enumerated), `nat` (recursive, no parameter), `option`, `prod`, `sum`, `sigma` (non recursive, with parameters), `list` (recursive with parameters), `eq` / `paths` (non recursive, with parameter and index) and `vector` (recursive with parameter and index). These examples strongly suggest that this generation could be automated.

Using Trocq in a context featuring axioms is possible but classical axioms should be handled with care. Indeed, Trocq is conservative over raw parametricity, and some classical axioms are known not to be parametric. For instance, postulating the existence of a (raw) parametricity translation for the Law of Excluded Middle is inconsistent. This means that one cannot hope in general to use Trocq on statements that mention classical axioms explicitly. Specific cases might work however: if they only occur in proofs but not in statements, or if definitions featuring such axioms remain parametric. From our experience, this remark makes Trocq useful in most scenarios, even when formalizing classical mathematics.

The concrete output of this work is a plugin [22] that consists of about 3000 l. of Rocq/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 [23, 28], generalized rewriting [50], and proof transfer [53]. Second, it addresses use cases, such as those of Section 8.2, that are beyond the skills of any existing tool in any proof assistant based on type theory. The logic programming paradigm provided by the Elpi meta-language proved perfectly well-suited for implementing this nature of proof search. The prototype plugin however arguably still 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 [49].

Acknowledgments

The authors are extremely grateful to András Kovács, Kenji Maillard, Enrico Tassi, Quentin Vermande, Théo Winterhalter, to anonymous members of the ESOP 2024 program committee and to the anonymous reviewers of the present submission for their helpful comments and suggestions.

References

[1] Reynald Affeldt and Cyril Cohen. 2023. Measure Construction by Extension in Dependent Type Theory with Application to Integration. arXiv:2209.02345 [cs.LO] accepted for publication in JAR.

[2] Reynald Affeldt, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, and Kazuhiko Sakaguchi. 2020. Competing inheritance paths in dependent type theory: a case study in functional analysis. In *IJCAR 2020 - International Joint Conference on Automated Reasoning*. Paris, France, 1–19. doi:10.1007/978-3-030-51054-1_1

[3] Xavier Allamigeon, Quentin Canu, and Pierre-Yves Strub. 2023. A Formal Disproof of Hirsch Conjecture. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023*, Robbert Krebbers, Dmitriy Traytel, Brigitte Pientka, and Steve Zdancewic (Eds.). ACM, 17–29. doi:10.1145/3573105.3575678

[4] Thorsten Altenkirch, Yorgo Chamoun, Ambrus Kaposi, and Michael Shulman. 2024. Internal Parametricity, without an Interval. *Proc. ACM Program. Lang.* 8, POPL (2024), 2340–2369. doi:10.1145/3632920

[5] Thorsten Altenkirch and Ambrus Kaposi. 2015. Towards a Cubical Type Theory without an Interval. In *TYPES (LIPIcs, Vol. 69)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 3:1–3:27. doi:10.4230/LIPIcs.TYPES.2015.3

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

[7] Carlo Angiuli, Guillaume Brunerie, Thierry Coquand, Robert Harper, Kuen-Bang Hou (Favonia), and Daniel R. Licata. 2021. Syntax and models of Cartesian cubical type theory. *Math. Struct. Comput. Sci.* 31, 4 (2021), 424–468. doi:10.1017/S0960129521000347

[8] Carlo Angiuli, Evan Cavallo, Anders Mörtberg, and Max Zeuner. 2021. Internalizing representation independence with univalence. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30. doi:10.1145/3434293

[9] David Aspinall and Adriana B. Compagnoni. 2001. Subtyping dependent types. *Theor. Comput. Sci.* 266, 1-2 (2001), 273–309. doi:10.1016/S0304-3975(00)00175-4

[10] Gilles Barthe, Venanzio Capretta, and Olivier Pons. 2003. Setoids in type theory. *J. Funct. Program.* 13, 2 (2003), 261–293. doi:10.1017/S0956796802004501

[11] Gilles Barthe and Olivier Pons. 2001. Type Isomorphisms and Proof Reuse in Dependent Type Theory. In *Foundations of Software Science and Computation Structures*, Furio Honsell and Marino Miculan (Eds.). Springer Berlin Heidelberg,

Berlin, Heidelberg, 57–71.

[12] 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, Paris, France, January 16-17, 2017*, Yves Bertot and Viktor Vafeiadis (Eds.). ACM, 164–172. doi:10.1145/3018610.3018615

[13] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. 2012. Proofs for free - Parametricity for dependent types. *J. Funct. Program.* 22, 2 (2012), 107–152. doi:10.1017/S0956796812000056

[14] Jean-Philippe Bernardy and Marc Lasson. 2011. Realizability and Parametricity in Pure Type Systems. In *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)*, Martin Hofmann (Ed.). Springer, 108–122. doi:10.1007/978-3-642-19805-2_8

[15] Jean-Philippe Bernardy and Guilhem Moulin. 2012. A Computational Interpretation of Parametricity. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*. IEEE Computer Society, 135–144. doi:10.1109/LICS.2012.25

[16] Sandrine Blazy. 2024. On Mechanized Semantics for Verified Compilation. Interview at the occasion of her invited talk at the ESOP 2024 conference. https://etaps.org/blog/017-sandrine-blazy/.

[17] Valentin Blot, Denis Cousineau, Enzo Crance, Louise Dubois de Prisque, Chantal Keller, Assia Mahboubi, and Pierre Vial. 2023. Compositional Pre-processing for Automated Reasoning in Dependent Type Theory. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023*, Robbert Krebbers, Dmitriy Traytel, Brigitte Pientka, and Steve Zdancewic (Eds.). ACM, 63–77. doi:10.1145/3573105.3575676

[18] Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. 2017. The next 700 syntactical models of type theory. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*, Yves Bertot and Viktor Vafeiadis (Eds.). ACM, 182–194. doi:10.1145/3018610.3018620

[19] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2018. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. In *21st International Conference on Types for Proofs and Programs (TYPES 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 69)*, Tarmo Uustalu (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 5:1–5:34. doi:10.4230/LIPIcs.TYPES.2015.5

[20] Cyril Cohen, Enzo Crance, and Assia Mahboubi. 2024. Artifact Report: Trocq: Proof Transfer for Free, With or Without Univalence. In *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 14576)*, Stephanie Weirich (Ed.). Springer, 269–274. doi:10.1007/978-3-031-57262-3_11

[21] Cyril Cohen, Enzo Crance, and Assia Mahboubi. 2024. Trocq: Proof Transfer for Free, With or Without Univalence. In *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 14576)*, Stephanie Weirich (Ed.). Springer, 239–268. doi:10.1007/978-3-031-57262-3_10

[22] Cyril Cohen, Enzo Crance, and Assia Mahboubi. 2025. *rocq-community/trocq: Trocq 0.1.8+prop.* doi:10.5281/zenodo.14846664

[23] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. 2013. Refinements for Free!. In *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings (Lecture Notes in Computer Science, Vol. 8307)*, Georges Gonthier and Michael Norrish (Eds.). Springer, 147–162. doi:10.1007/978-3-319-03545-1_10

[24] Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. 2020. Hierarchy Builder: algebraic hierarchies made easy in Coq with Elpi. In *FSCD 2020 - 5th International Conference on Formal Structures for Computation and Deduction*. Paris, France, 34:1–34:21. doi:10.4230/LIPIcs.FSCD.2020.34

[25] Thierry Coquand and Gérard P. Huet. 1988. The Calculus of Constructions. *Inf. Comput.* 76, 2/3 (1988), 95–120. doi:10.1016/0890-5401(88)90005-3

[26] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12699)*, André Platzer and Geoff Sutcliffe (Eds.). Springer, 625–635. doi:10.1007/978-3-030-79876-5_37

[27] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.).

ACM, 689–700. doi:10.1145/2676726.2677006

[28] Maxime Dénès, Anders Mörtberg, and Vincent Siles. 2012. A Refinement-Based Approach to Computational Algebra in Coq. In *Interactive Theorem Proving*, Lennart Beringer and Amy Felty (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 83–98. doi:10.1007/978-3-642-32347-8_7

[29] Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. 2015. ELPI: Fast, Embeddable, λProlog Interpreter. In *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)*, Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov (Eds.). Springer, 460–468. doi:10.1007/978-3-662-48899-7_32

[30] Thom Frühwirth and Frank Raiser. 2011. Constraint Handling Rules: Compilation, Execution, and Analysis.

[31] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. 2019. Definitional proof-irrelevance without K. *Proc. ACM Program. Lang.* 3, POPL (2019), 3:1–3:28. doi:10.1145/3290316

[32] Kiran Gopinathan, Mayank Keoliya, and Ilya Sergey. 2023. Mostly Automated Proof Repair for Verified Libraries. *Proc. ACM Program. Lang.* 7, PLDI (2023), 25–49. doi:10.1145/3591221

[33] Sébastien Gouëzel. 2021. Vitali-Carathéodory theorem in mathlib. https://leanprover-community.github.io/mathlib_docs/measure_theory/integral/vitali_caratheodory.html.

[34] Florian Haftmann, Alexander Krauss, Ondřej Kunčar, and Tobias Nipkow. 2013. Data Refinement in Isabelle/HOL. In *Interactive Theorem Proving*, Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 100–115.

[35] Brian Huffman and Ondřej Kunčar. 2013. Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL. In *Certified Programs and Proofs*, Georges Gonthier and Michael Norrish (Eds.). Springer International Publishing, Cham, 131–146.

[36] Chantal Keller and Marc Lasson. 2012. Parametricity in an Impredicative Sort. In *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)*, Patrick Cégielski and Arnaud Durand (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 381–395. doi:10.4230/LIPIcs.CSL.2012.381

[37] Neelakantan R. Krishnaswami and Derek Dreyer. 2013. Internalizing Relational Parametricity in the Extensional Calculus of Constructions. In *Computer Science Logic 2013 (CSL 2013) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 23)*, Simona Ronchi Della Rocca (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 432–451. doi:10.4230/LIPIcs.CSL.2013.432

[38] Peter Lammich. 2013. Automatic Data Refinement. In *Interactive Theorem Proving*, Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 84–99.

[39] Peter Lammich and Andreas Lochbihler. 2019. Automatic Refinement to Efficient Data Structures: A Comparison of Two Approaches. *J. Autom. Reason.* 63, 1 (2019), 53–94. doi:10.1007/s10817-018-9461-9

[40] Théo Laurent, Meven Lennon-Bertrand, and Kenji Maillard. 2024. Definitional Functoriality for Dependent (Sub)Types. In *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 14576)*, Stephanie Weirich (Ed.). Springer, 302–331. doi:10.1007/978-3-031-57262-3_13

[41] Nicolas Magaud. 2003. Changing Data Representation within the Coq System. In *TPHOLs'2003*, Vol. 2758. LNCS, Springer-Verlag. http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=2758&spage=87 © Springer-Verlag.

[42] Érik Martin-Dorel and Guillaume Melquiond. 2016. Proving Tight Bounds on Univariate Expressions with Elementary Functions in Coq. *J. Autom. Reason.* 57, 3 (2016), 187–217. doi:10.1007/s10817-015-9350-4

[43] The mathlib Community. 2020. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, Jasmin Blanchette and Catalin Hritcu (Eds.). ACM, 367–381. doi:10.1145/3372885.3373824

[44] John C. Mitchell. 1986. 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.* ACM Press, 263–276. doi:10.1145/512644.512669

[45] Rob Nederpelt and Herman Geuvers. 2014. *Type Theory and Formal Proof: An Introduction.* Cambridge University Press. doi:10.1017/CBO9781139567725

[46] Ulf Norell. 2008. Dependently Typed Programming in Agda. In *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures (Lecture Notes in Computer Science, Vol. 5832)*, Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra (Eds.). Springer, 230–266. doi:10.1007/978-3-642-04652-0_5

[47] 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. College Publications. https://inria.hal.science/hal-01094195

[48] John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, R. E. A. Mason (Ed.). North-Holland/IFIP, 513–523.

[49] Talia Ringer, RanDair Porter, Nathaniel Yazdani, John Leo, and Dan Grossman. 2021. Proof repair across type equivalences. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 112–127. doi:10.1145/3453483.3454033

[50] Matthieu Sozeau. 2009. A New Look at Generalized Rewriting in Type Theory. *J. Formaliz. Reason.* 2, 1 (2009), 41–62. doi:10.6092/issn.1972-5787/1574

[51] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. 2020. The MetaCoq Project. *J. Autom. Reason.* 64, 5 (2020), 947–999. doi:10.1007/s10817-019-09540-0

[52] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. 2018. Equivalences for free: univalent parametricity for effective transport. *Proc. ACM Program. Lang.* 2, ICFP, Article 92 (jul 2018), 29 pages. doi:10.1145/3236787

[53] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. 2021. The Marriage of Univalence and Parametricity. *J. ACM* 68, 1, Article 5 (jan 2021), 44 pages. doi:10.1145/3429979

[54] Enrico Tassi. 2019. 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. doi:10.4230/LIPIcs.CVIT.2016.23

[55] The Coq Development Team. 2024. *The Coq Proof Assistant.* doi:10.5281/zenodo.14542673

[56] The Rocq Development Team. 2025. *The Rocq Prover.* doi:10.5281/zenodo.15149629

[57] The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study.

[58] Antoine Van Muylder, Andreas Nuyts, and Dominique Devriese. 2024. Internal and Observational Parametricity for Cubical Agda. *Proc. ACM Program. Lang.* 8, POPL (2024), 209–240. doi:10.1145/3632850

[59] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2019. Cubical agda: a dependently typed programming language with univalence and higher inductive types. *Proc. ACM Program. Lang.* 3, ICFP (2019), 87:1–87:29. doi:10.1145/3341691

[60] Cosmo Viola, Max Fan, and Talia Ringer. 2024. Proof Repair across Quotient Type Equivalences. (2024). https://arxiv.org/abs/2310.06959v5.

[61] Philip Wadler. 1989. Theorems for Free!. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, Joseph E. Stoy (Ed.). ACM, 347–359. doi:10.1145/99370.99404

[62] Conrad Watt, Maja Trela, Peter Lammich, and Florian Märkl. 2023. WasmRef-Isabelle: A Verified Monadic Interpreter and Industrial Fuzzing Oracle for WebAssembly. *Proc. ACM Program. Lang.* 7, PLDI (2023), 100–123. doi:10.1145/3591224

[63] Yu Zhang, Jérémie Koenig, Zhong Shao, and Yuting Wang. 2025. Unifying Compositional Verification and Certified Compilation with a Three-Dimensional Refinement Algebra. *Proc. ACM Program. Lang.* 9, POPL (2025), 1903–1933. doi:10.1145/3704900

[64] Théo Zimmermann and Hugo Herbelin. 2015. Automatic and Transparent Transfer of Theorems along Isomorphisms in the Coq Proof Assistant. In *Conference on Intelligent Computer Mathematics*. Washington, D.C., United States. https://hal.science/hal-01152588