Trocq: Proof Transfer for Free, With or Without Univalence

Cyril Cohen, Enzo Crance, Assia Mahboubi

Trocq: Proof Transfer for Free, With or Without Univalence

CYRIL COHEN*, Université Côte d'Azur, Inria, France
ENZO CRANCE*, Mitsubishi Electric R&D Centre Europe, France and Nantes Université, École Centrale Nantes, CNRS, INRIA, LS2N, UMR 6004, France
ASSIA MAHBOUBI*, Nantes Université, École Centrale Nantes, CNRS, INRIA, LS2N, UMR 6004, France

In interactive theorem proving, a range of different representations may be available for a single mathematical concept, and some proofs may rely on several representations. Without automated support such as proof transfer, theorems available with different representations cannot be combined, without light to major manual input from the user. Tools with such a purpose exist, but in proof assistants based on dependent type theory, it still requires human effort to prove transfer, whereas it is obvious and often left implicit on paper.

This paper presents Trocq, a new proof transfer framework, based on a generalization of the univalent parametricity translation, thanks to a new formulation of type equivalence. This translation takes care to avoid dependency on the axiom of univalence for transfers in a delimited class of statements, and may be used with relations that are not necessarily isomorphisms. We motivate and apply our framework on a set of examples designed to show that it unifies several existing proof transfer tools. The article also discusses an implementation of this translation for the Coq proof assistant, in the Coq-Elpi meta-language.

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

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

1 INTRODUCTION

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

Example 1.1 (Relating proof-oriented data-structures with computation-oriented ones). The standard library of the Coq proof assistant [The Coq Development Team 2022] actually proposes two data structures for representing natural numbers. Type $\mathbb{N}$ uses a unary representation, so that the associated elimination principle `ℕ_ind` expresses the usual recurrence scheme:

```
Inductive ℕ :=
  | 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.
```

*All authors contributed equally to this research.

Authors' addresses: Cyril Cohen, cyril.cohen@inria.fr, Université Côte d'Azur, Inria, 2004 route des Lucioles, Valbonne, France, 06902; Enzo Crance, Mitsubishi Electric R&D Centre Europe, Rennes, France, 35700 and Nantes Université, École Centrale Nantes, CNRS, INRIA, LS2N, UMR 6004, Nantes, France, 44300; Assia Mahboubi, AssiaMahboubi, Nantes Université, École Centrale Nantes, CNRS, INRIA, LS2N, UMR 6004, Nantes, France, 44000-F.

Type `N` uses a binary representation `positive` of non-negative integers, as sequences of bits with a head 1, and is thus better suited for coding efficient arithmetic operations. The successor function $S_N : N \to N$ 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 : Set :=
  | xI : positive → positive (* p1 *)
  | xO : positive → positive (* p0 *)
  | xH : positive.           (* 1  *)

Inductive N : Set :=
  | 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 is useful to implement conversions $\uparrow_{\mathbb{N}} : N \to \mathbb{N}$ and $\downarrow_{\mathbb{N}} : \mathbb{N} \to N$ between the unary and binary representations. These conversion functions are in fact inverses of each other. The natural recurrence scheme on natural numbers thus *transfers* to type `N`:

```
N_ind : ∀ P : N → □, P O_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 compatibility lemma:

$$\forall n : \mathbb{N}, \quad \downarrow_{\mathbb{N}} (S_{\mathbb{N}}\ n) = S_N\ (\downarrow_{\mathbb{N}}\ n)$$

Program verification supplies numerous examples of proof transfer use-cases, but this issue goes way beyond computational concerns. For instance, the formal study of summation and integration, in basic real analysis, provides a classic example of frustrating proof transfer bureaucracy.

Example 1.2 (Extended domains). 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 point-wise addition of u and v, is also a summable sequence, and that:

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

Making the definition of the real number $\sum u$ depend on a summability witness does not scale, as every other algebraic operation "under the sum" then requires a new proof of summability. In a classical setting, the standard approach rather assigns a default value to the case of an infinite sum, for instance by introducing an extended domain $[0, +\infty]$, and extending the addition operation 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 prove first that Equation 1 holds for *any* two sequences of *extended* non-negative numbers. The result is then *transferred* to the special case of summable sequences of non-negative numbers. Major libraries of formalized mathematics including Lean's mathlib [DBL 2020], Isabelle/HOL's Archive of Formal Proofs, coq-interval [Martin-Dorel and Melquiond 2016] or Coq's mathcomp-analysis [Affeldt and Cohen 2023], resort to such extended domains and transfer steps, notably for defining measure theory. Yet, as reported by expert users [Gouëzel 2021], the associated transfer

bureaucracy is essentially done manually and thus significantly clutters formal developments in real and complex analysis, probabilities, etc.

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

In this paper, we focus on interactive theorem provers based on dependent type theory, such as Coq, Agda [Norell 2008] or Lean [de Moura and Ullrich 2021]. 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 [Mitchell 1986] and parametricity meta-theorems [Reynolds 1983; Wadler 1989]. 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 the proof of parametricity properties for type families and for programs. Such syntactic relational models can accommodate dependent types [Bernardy and Lasson 2011], inductive types [Bernardy et al. 2012] and in fact the full Calculus of Inductive Constructions, including its impredicative sort [Keller and Lasson 2012].

In particular, the *univalent parametricity* translation [Tabareau et al. 2021] makes benefit of the univalence axiom [Univalent Foundations Program 2013] so as to transfer programs and theorems 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 invariance principle* provides internal representation independence results, for more general relational correspondences between types than equivalences [Angiuli et al. 2021]. This last approach is thus particularly relevant in the frame of cubical type theory [Cohen et al. 2017; Vezzosi et al. 2019]. Indeed, a computational interpretation of the univalence axiom brings computational adequacy to otherwise possibly stuck terms, those resulting from a transfer involving an axiomatized univalence principle.

Unfortunately, a Swiss-army knife for automating the bureaucracy of proof transfer is still missing from the arsenal available to users of major proof assistants like Coq, Lean or Agda. Besides implementation concerns, the above examples actually illustrate fundamental limitations to the scope of existing approaches:

Univalence is overkill. Both univalent parametricity and the structure invariance principle can be used to derive the statement and the proof of the induction principle `N_ind` of Example 1.1, from the elimination scheme of type $\mathbb{N}$. But up to our knowledge, all the existing methods for automating this implication will pull in the univalence principle in the proof, although it can be obtained by hand by very elementary means. This is all the more frustrating that the univalence axiom is incompatible with proof irrelevance, a commonly assumed axiom in libraries formalizing classical mathematics, as Lean's mathlib.

Equivalences are not enough, neither are quotients. Univalent parametricity cannot help with our Example 1.2, as it is geared towards equivalences. But in this case, we are in fact not aware of an implemented method which would apply. In particular, the structure invariance principle does not apply as such in such an instance of quasi-PER [Krishnaswami and Dreyer 2013].

This leads us to the crux of our problem: existing techniques for transferring results from one type to another, *e.g.*, from $\mathbb{N}$ to `N` or from extended real numbers to real numbers, are either not suitable for dependent types, or too coarse to track the exact amount of data needed in a given proof, and not more.

Contributions. This paper presents three contributions:

- A parametricity framework *à la carte*, which generalizes [Tabareau et al. 2021]'s univalent parametricity translation, as well as refinements à la CoqEAL [Cohen et al. 2013] or generalized rewriting [Sozeau 2009]. Its pivotal ingredient is an appropriate, and up to our knowledge novel, phrasing of type equivalence, which allows for a finer-grained control of the data propagated by the translation.
- A conservative subtyping extension of CC_ω [Coquand and Huet 1988], used to formulate an inference algorithm for the synthesis of parametricity proofs.
- The implementation of a new parametricity plugin for the Coq proof assistant, using the Coq-Elpi [Tassi 2019] meta-language. This plugin is based on original supporting formal proofs, conducted on top of the HoTT library [Bauer et al. 2017], and distributed with a collection of application examples.

Outline. The rest of this paper is organized as follows. Section 2 introduces proof transfer and recalls the principle, strengths and weaknesses of the univalent parametricity translation. In Section 3, we present a new definition of type equivalence and we put this definition to good use in a hierarchy of structures for relations preserved by parametricity. Section 4 then presents a stratified variant of the univalent parametricity translation. In Section 5, we introduce the technological and programming choices that drive the implementation of the companion artifact, starting with a short description of plugin making with Coq-Elpi, and continuing with design choices. In Section 6, we eventually discuss a few applications, including Examples and 1.1 and 1.2, before concluding in Section 7.

2 STRENGTHS AND LIMITS OF UNIVALENT PARAMETRICITY

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

2.1 Proof transfer in type theory, in practice

Let us first recall the syntax of the Calculus of Constructions, CC_ω, a *lambda*-calculus with dependent function types and a predicative hierarchy of universes, denoted $\square_i$:

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

We omit the typing rules of the calculus, available in standard presentations [Nederpelt and Geuvers 2014]. We will also use the standard equality type, called propositional equality, as well as dependent pairs, denoted $\Sigma x : A.\, B$. Proof assistants Coq, Agda and Lean are based on various extensions of this core, notably with inductive types and with an impredicative sort. When the universe level does not matter, we will casually remove the annotation and use notation $\square$.

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

$$\Gamma \vdash w : \forall (t_1 : T_1)(t_2 : T_2), R_T\ t_1\ t_2 \to R_\square (P\ t_1)(Q\ t_2)$$

for a suitable context Γ. This setting generalizes as expected to k-ary type formers, and to more pairs of related types. In practice, relation $R_\square$ is often a right-to-left arrow, *i.e.*, $R_\square\ A\ B \triangleq B \to A$, as in this case the proof w substantiates a proof step turning goal clause $\Gamma \vdash P\ t_1$ into $\Gamma \vdash Q\ t_2$. Phrased as such, this synthesis problem is quite loosely specified, and some solutions are in fact too trivial to be of interest. Consider for instance the case of a *functional* relation between T_2 and T_1, with $R_T\ t_1\ t_2$ defined as $t_1 = \phi\ t_2$, for some $\phi : T_2 \to T_1$. In this case, the composition $P \circ \phi$ is an obvious solution Q, but often an uninformative one. Indeed, this composition can only propagate structural arguments, but does not incorporate additional mathematical proofs of program equivalences, potentially available in the context. For instance, going back to Example 1.1, both statements `Q_trivial` and `Q_ok` from below can possibly be synthesized from `P`. The latter is more informative, but can only be obtained if proofs that relate $\mathsf{S}_\mathbb{N}$ with S_N and $\mathsf{O}_\mathbb{N}$ with O_N and are available.

```
P (n : ℕ) : S_ℕ n ≠ O_ℕ.

Q_trivial (n : N) : S_ℕ (↑_ℕ n) ≠ O_ℕ.
Q_ok (n : N) : S_N n ≠ O_N.
```

Automation devices dedicated to proof transfer thus typically consist of a meta-program which attempts to compute the type former Q and the proof w by induction on the structure of P, by composing registered canonical pairs of related terms, and the corresponding proofs. These tools differ by the nature of relations they can accommodate, and by the class of type formers they are able to synthesize. For instance, *generalized rewriting* [Sozeau 2009], which provides essential support to formalizations based on setoids [Barthe et al. 2003], addresses the case of homogeneous (and reflexive) relations, *i.e.*, when T_1 and T_2 coincide. The CoqEAL library [Cohen et al. 2013] provides another example of such transfer automation tool, geared towards *refinements*, typically from a proof-oriented data-structure to a computation-oriented one. It is thus specialized to heterogeneous, functional relations but restricted to closed, quantifier-free type formers. We now discuss the few transfer methods which can accommodate dependent types and heterogeneous relations.

2.2 Type equivalences, univalence

Let us first focus on the special case of types related by an *equivalence*, and start with a few standard definitions, notations and lemmas. Omitted details can be found in the usual references, like the Homotopy Type Theory book [Univalent Foundations Program 2013]. Two functions $f, g : A \to B$ are *point-wise equal*, denoted $f \doteq g$ when their values coincide on all arguments, that is $f \doteq g : \Pi a : A.\ f\ a = g\ a$. For any type A, id_A denotes $\lambda a : A.\ a$, the identity function on A, and we will write id when the implicit type A is not ambiguous.

Definition 2.1 (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, which respectively assert that g is both a pointwise left and right inverse of f. An isomorphism f is an *equivalence*, denoted $\mathsf{IsEquiv}(f)$, when it moreover enjoys a last *adjunction property*, relating the proofs of the section and retraction properties and ensuring that $\mathsf{IsEquiv}(f)$ is proof-irrelevant.

Two 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 2.2. *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 indistinguishable.

Definition 2.3 (Univalence principle). For any two types A and B, the canonical map $A = B \to A \simeq B$ is an equivalence.

In variants of CC_ω, the univalence principle can be postulated as an axiom, with no explicit computational content, as done for instance in the HoTT library for the Coq proof assistant [Bauer et al. 2017]. Some more recent variants of dependent type theory feature a built-in computational univalence principle, and are used to implement experimental proof assistants, such as Cubical Agda. 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 [Ringer et al. 2021].

Going back to our example from § 2.1, transferring along an equivalence $\mathbb{N} \simeq \mathsf{N}$ will thus produce `Q_trivial` from `P`. In presence of a computational univalence principle, the structure identity principle can be put to good use for deriving the more informative `Q_ok`, at the cost of requiring an explicit interface for the signature to be transported, thus a successor function in our case [Angiuli et al. 2021]. In the case of an axiomatic univalent principle, it is also possible to achieve this transport from `P` to `Q_ok`, using a method called *univalent parametricity* [Tabareau et al. 2021], which we shall discuss in the next section.

2.3 Parametricity translations

Univalent parametricity strengthens the transfer principle provided by the univalence axiom by combining it with parametricity. In CC_ω, the essence of parametricity, which is to devise a relational interpretation of types, can be turned into an actual syntactic translation, as relations can themselves be modeled as λ-terms in CC_ω. The seminal work of Bernardy *et al.*, Keller and Lasson combines in what we refer to as a *raw parametricity translation*, which essentially defines inductively a logical relation $[\![T]\!]$ for any type T, as described on Figure 1.

- Context translation:

$$[\![\langle\rangle]\!] = \langle\rangle \tag{2}$$

$$[\![\Gamma, x : A]\!] = [\![\Gamma]\!], x : A, x' : A', x_R : [\![A]\!]\ x\ x' \tag{3}$$

- Term translation:

$$[\![\square_i]\!] = \lambda A\, A'.\, A \to A' \to \square_i \tag{4}$$

$$[\![x]\!] = x_R \tag{5}$$

$$[\![\Pi x : A.\, B]\!] = \lambda f\, f'.\, \Pi(x : A)(x' : A')(x_R : [\![A]\!]\ x\ x').\, [\![B]\!](f\ x)(f'\ x') \tag{6}$$

$$[\![\lambda x : A.\, t]\!] = \Pi(x : A)(x' : A')(x_R : [\![A]\!]\ x\ x').\, [\![t]\!] \tag{7}$$

$$[\![A\ B]\!] = [\![A]\!]\ B\ B'\ [\![B]\!] \tag{8}$$

Fig. 1. Raw parametricity translation for CC_ω.

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. The associated abstraction theorem ensures that this translation preserves typing, in the following sense:

Theorem 2.4. *If* $\Gamma \vdash t : T$ *then* $[\![\Gamma]\!] \vdash t : T$, $[\![\Gamma]\!] \vdash t' : T'$ *and* $[\![\Gamma]\!] \vdash [\![t]\!] : [\![T]\!]\ t\ t'$.

Proof. See for instance [Keller and Lasson 2012]. □

This translation precisely generates the statements expected from a parametric type family or program. For instance, the translation of a Π-type, given by Equation 6, is a type of relations on functions, which relates those producing related outputs from related inputs. Concrete implementations of this translation are available [Keller and Lasson 2012; Tassi 2019], and useful to generate and prove parametricity properties for type families or for constants, improved induction schemes, etc.

The key observation of univalent parametricity is that, it is possible to preserve the abstraction theorem while restricting to relations that are in fact (heterogeneous) equivalences. This however requires a careful design in the translation of universes:

$$[\![\Box_i]\!]\ A\ B \quad \triangleq \quad \Sigma(R : A \to B \to \Box_i)(e : A \simeq B).\, \Pi(a : A)(b : B).\, R\ a\ b \simeq (a = \downarrow_e b)$$

where $[\![\cdot]\!]$ now refers to the univalent parametricity translation, replacing the notation introduced for the raw variant. This type packages a relation R and an equivalence e such that R is equivalent to the functional relation associated with $\downarrow_e$. Crucially, one can show that under the univalence axiom, $[\![\Box_i]\!]$ is equivalent to equivalence, that is for any two types A and B, we have:

$$[\![\Box_i]\!]\ A\ B \simeq (A \simeq B).$$

This observation is actually an instance of a more general technique available for constructing syntactic models of type theory [Boulier et al. 2017]. In these models, a standard way to recover the abstraction theorem consists in refining the translation into two variants, for terms $T : \Box_i$, that are also types. Its translation as a *term*, denoted $[\,T\,]$, should be a dependent pair, which equips a relation with the additional data prescribed by the interpretation $[\![\Box_i]\!]$ of the universe. The translation $[\![T]\!]$ of T as a *type* will be the relation itself, that is, the projection of the dependent pair $[\,T\,]$ onto its first component, denoted $\mathsf{rel}([\,T\,])$. We refer to the original publication [Tabareau et al. 2021, Figure 4] for a complete description of the translation.

We can now phrase the resulting abstraction theorem [Tabareau et al. 2021], where $\vdash_u$ refers to a typing judgment assuming the univalence axiom:

Theorem 2.5. *If* $\Gamma \vdash t : T$ *then* $[\![\Gamma]\!] \vdash_u [\,t\,] \ :\ [\![T]\!]\ t\ t'$.

Note that proving the abstraction theorem 2.5 involves in particular proving that:

$$[\,\Box_i\,] \ :\ [\![\Box_{i+1}]\!]\ \Box_i\ \Box_i.$$

As a consequence, the definition of relation $[\,\Box_i\,]$ uses the univalence principle in an essential way, in order to prove that the relation in the universe is equivalent to equality on the universe, *i.e.*, to prove that:

$$\Pi A\, B : \Box_i.\, [\![\Box_i]\!]\ A\ B \simeq (A = B).$$

Crucially this univalent parametricity translation can be seamlessly extended so as to also make use of a global context of user-defined equivalences. Now let us go back to our motivating Example 1.1. A closer look at [Tabareau et al. 2021, Figure 4] reveals why the univalent parametricity translation can only resort to the univalence axiom in transferring the recurrence principle from type $\mathbb{N}$ to type `N`. Because of the above remark, on the abstraction theorem for universes, univalence will actually be necessary as soon as the translated term involves an essential occurrence of a universe $\Box_i$.

3 TYPE EQUIVALENCE IN KIT

In this section, we rephrase and split type equivalence into pieces. First in Section. 3.1 we give an equivalent presentation of type equivalence as a nested sigma type. Then in Section. 3.2, we carve a hierarchy of relations out of this nested sigma type by selectively picking pieces.

3.1 Disassembling type equivalence

Let us first observe that the Definition 2.1, of type equivalence is quite asymmetrical, although this fact is somehow put under the rug by the infix $A \simeq B$ notation. First, the data of an equivalence $e : A \simeq B$ privilege 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 adjunction property, which we eluded in Definition 2.1, 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 provides an equivalent definition. This entanglement prevents any hope to trace 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 [Univalent Foundations Program 2013] however suggests a symmetrical wording of the definition of type equivalence, in terms of functional relations.

Definition 3.1. Any relation $R : A \to B \to \square_i$, is *functional*, denoted $\mathsf{IsFun}(R)$, 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 $\Sigma t : T.\, \Pi t' : T.\, t = t'$.

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

Lemma 3.2. *For any types $A, B : \square_i$, the type $A \simeq B$ is equivalent to:*

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

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

Let us sketch a proof of this result, left as an exercise in [Univalent Foundations Program 2013]. We need the following lemma, which explains why $\mathsf{IsFun}()$ characterizes functional relations:

Lemma 3.3. *For any types $A, B : \square_i$, we have* $(A \to B) \simeq \Sigma R : A \to B \to \square_i.\ \mathit{IsFun}(R)$.

Proof. The proof goes by chaining the following equivalences:

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

□

Proof of Lemma 3.2. The proof goes by chaining the following equivalences:

$$\begin{array}{lll}
(A \simeq B) & \simeq \Sigma f : A \to B.\, \mathsf{IsEquiv}(f) & \text{by definition of } (A \simeq B) \\
& \simeq \Sigma f : A \to B.\, \Pi b : B.\, \mathsf{IsContr}(\Sigma a.f\ a = b) & \text{standard result in HoTT} \\
& \simeq \Sigma f : A \to B.\, \mathsf{IsFun}(\lambda(b : B)(a : A).\, f\ a = b) & \text{by definition of } \mathsf{IsFun}(\cdot) \\
& \simeq \Sigma\left(f : \Sigma R : A \to B \to \square_i.\, \mathsf{IsFun}(R)\right).\, \mathsf{IsFun}(\pi_1(f)^{-1}) & \text{by Lemma 3.3} \\
& \simeq \Sigma R : A \to B \to \square_i.\, \mathsf{IsFun}(R) \times \mathsf{IsFun}(R^{-1}) & \text{by associativity of } \Sigma
\end{array}$$

□

The symmetrical version of type equivalence provided by Lemma 3.2 however 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. The essential ingredient of this rewording is the alternative characterization of functional relations.

Definition 3.4. For any types $A, B : \square_i$, a relation $R : A \to B \to \square_i$, is a *univalent map*, denoted $\mathsf{IsUmap}(R)$ when there exists a function $m : A \to B$ together with proofs $g_1 : \Pi(a : A)(b : B).\, m\ a = b \to R\ a\ b$ and $g_2 : \Pi(a : A)(b : B).\, R\ a\ b \to m\ a = b$ such that:

$$\Pi(a : A)(b : B).\, (g_1\ a\ b) \circ (g_2\ a\ b) \doteqdot id.$$

Now comes the crux lemma of this section, formally proved in the companion code.

Lemma 3.5. *For any types $A, B : \square_i$ and any relation $R : A \to B \to \square_i$*

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

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

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

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

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

which proof we do not detail, referring the reader to the companion code. □

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

Theorem 3.6. *For any types $A, B : \square_i$, we have:*

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

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

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

The resulting collection of data is now symmetrical, as the reverse direction of the equivalence based on univalent maps can be obtained by flipping the relation and swapping the two functionality proofs. If the η-rule for records is verified, symmetry is even definitionally involutive.

3.2 Reassembling type equivalence

Theorem 3.6 obviously ensures that $\boxdot^{\top}$ is an equivalence relation. but the pivotal observation for the rest of this paper is the following: it is possible to prove the relation at the universe, that is, to construct a term of type $\boxdot^{\top}\ \square_i\ \square_i$, which uses the univalence axiom *only* for the two last pieces of data, that is, when proving that the relation is included in the graph of the map, and when proving the coherence property.

The rest of the paper is thus devoted to the study and implementation of an improved variant of Tabareau *et al.*'s univalent parametricity translation, in which the interpretation of universes is blown up into a collection of relation structures. We thus introduce a collection of relations on types, obtained by gradually weakening the structure of equivalence relation.

Definition 3.7. For $n, k \in \{0, 1, 2\mathrm{a}, 2\mathrm{b}, 3, 4\}$, and $\alpha = (n, k)$, we introduce $\boxdot^\alpha$, defined as:

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

where the *map class* $\mathrm{Class}_\alpha\ R$ itself unfolds to $(\mathsf{M}_n\ R) \times (\mathsf{M}_k\ R^{-1})$, with $\mathsf{M}_n : (A \to B \to \square) \to \square$ defined as:

$$\mathsf{M}_0\ R \triangleq .$$
$$\mathsf{M}_1\ R \triangleq (A \to B)$$
$$\mathsf{M}_{2\mathrm{a}}\ R \triangleq \Sigma m : A \to B.\, \Pi a\, b.\, m\ a = b \to R\ a\ b$$
$$\mathsf{M}_{2\mathrm{b}}\ R \triangleq \Sigma m : A \to B.\, \Pi a\, b.\, R\ a\ b \to m\ a = b$$
$$\mathsf{M}_3\ R \triangleq \Sigma m : A \to B.\, \Sigma(g_1 : \Pi a\, b.\, m\ a = b \to R\ a\ b).\, \Pi a\ b.\, R\ a\ b \to m\ a = b$$
$$\mathsf{M}_4\ R \triangleq \Sigma m : A \to B.\, \Sigma g_1.\, \Sigma(g_2 : \Pi a\, b.\, R\ a\ b \to m\ a = b).\, \Pi a\, b.\, (g_1\ a\ b) \circ (g_1\ a\ b) \doteqdot id$$

For any types A and B, and any $r : \boxdot^\alpha\ A\ B$ we will use notations $\mathrm{rel}(r)$, $\mathrm{map}(r)$ and $\mathrm{comap}(r)$ 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. The relation $\boxdot^{(4,4)}$ is also denoted $\boxdot^\top$, in accordance with the notation introduced in Theorem 3.6. Similarly, $\boxdot^\bot$ refers to $\boxdot^{(0,0)}$.

Strictly speaking all the constants introduced by definitions 3.7 are also parameterized by a universe level, but it was omitted here for the sake of readability. The corresponding lattice to the collection of M_n dependent tuples is depicted on Figure 2. Each arrow represents an inclusion of the data packed in the source tuple, into the data packed in the target one. In the code, these tuples are record types, and nodes on the figure are labeled with the names of the corresponding record fields introduced by the richer structure.

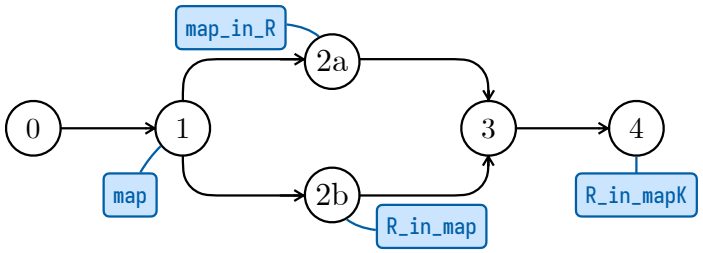

Fig. 2. Lattice of the stratified parametricity record hierarchy

3.3 The proofs of $\boxdot^\alpha\ \square\ \square$

Now that we have a proper definition for $\boxdot^\alpha$, the first question we ask is that of the proofs of $\boxdot^\alpha$. Indeed, in the raw translation we have $\vdash (\boxdot^\bot, (.\,,.)) : \boxdot^\bot\ \square\ \square$, and in the univalent translation there is a witness p such that $\vdash p : \boxdot^\top\ \square\ \square$. Now let us try to figure out what the equivalent of p should be in our system. Let us assume we have a translation $[\![\cdot]\!]$ as in the univalent translation. Given $\Gamma \vdash A : \square$, we would have both $[\![A]\!] : \boxdot^\alpha\ A\ A'$ but an abstraction theorem would give $[\![A]\!] : [\![\square]\!]\ A\ A'$ hence $[\![\square]\!] = \boxdot^\alpha$ for all α, which is not possible.

This means we need to update our term parametricity translation to take into account this annotation. As a second attempt, we define a notation $[\, A \mathbin{@} \square^{\alpha} \,]$ to direct the translation of A. But ig we apply it to $\square$, we get both

$$\begin{aligned}[\, \square \mathbin{@} \square^{\alpha} \,] &: \boxdot^{\alpha}\ \square\ \square \\ &: [\![\, \square \,]\!]\ \square\ \square \quad \text{by the abstraction theorem}\end{aligned}$$

which leads to the same impossibility. This means we need to systematically annotate sorts, so that the term translation of $\square$ now looks like $[\, \square^{\gamma} \mathbin{@} \square^{\alpha} \,]$. Let us check that we can indeed typecheck this correctly. We have

$$\begin{aligned}[\, \square^{\gamma} \mathbin{@} \square^{\alpha} \,] &: \boxdot^{\alpha}\ \square^{\gamma}\ \square^{\gamma} \\ &: [\![\, \square^{\alpha} \,]\!]\ \square^{\gamma}\ \square^{\gamma} \quad \text{by the abstraction theorem}\end{aligned}$$

Now we have $\mathsf{rel}([\, \square^{\alpha} \mathbin{@} \square^{\beta} \,]) = [\![\, \square^{\alpha} \,]\!]$, hence we must find $[\, \square^{\alpha} \mathbin{@} \square^{\beta} \,]$ verifying the two following constraints:

$$\begin{aligned}\mathsf{rel}([\, \square^{\alpha} \mathbin{@} \square^{\beta} \,]) &= [\![\, \square^{\alpha} \,]\!] \\ [\, \square^{\alpha} \mathbin{@} \square^{\beta} \,] &: \boxdot^{\beta}\ \square^{\alpha}\ \square^{\alpha}\end{aligned}$$

which is finally consistent.

We provide in the supplementary material[1] the terms $p_{\square}^{\alpha,\beta}$ such that $[\, \square^{\alpha} \mathbin{@} \square^{\beta} \,] = p_{\square}^{\alpha,\beta}$. We show $p_{\square}$ is well defined over a domain $\mathcal{D}_{\square} \subseteq \mathcal{A}^2$. We characterize the terms $p_{\square}^{\alpha,\beta}$ that rely on univalence, and we give an explicit description of $\mathcal{D}_{\square}$.

Theorem 3.8 (Univalence in $p_{\square}^{\alpha,\beta}$). *$p_{\square}^{\alpha,(m,n)}$ uses univalence if and only if $(m,n) \notin \{0,1,2a\}^2$.*

Theorem 3.9 (Characterization of $\mathcal{D}_{\square}$).

$$\begin{aligned}\mathcal{D}_{\square} = \{&(\top, (m,n)) \mid (m,n) \notin \{0,1,2a\}^2\} \\ \cup\, \{&(\alpha, (m,n)) \mid m,n \in \{0,1,2a\},\ \alpha \in \mathcal{A}\}\end{aligned}$$

3.4 Translation of dependent products

The core lemma used to translate dependent products combines parametricity proofs on the domain and co-domain of the product. In stratified parametricity, this lemma is p_{Π}^{γ} and is indexed by the desired parametricity class, which we provide in the supplementary material[2]. We also have a function $\mathcal{D}_{\Pi} : \mathcal{A} \to \mathcal{A}^2$, which tells us the minimal requirements $\mathcal{D}_{\Pi}(\gamma) = (\alpha, \beta)$ on the arguments A and B of $\Pi a : A.\, B\ a$ in order to get an element in $\boxdot^{\gamma}$.

We observe that we can define an operator $\rtimes$, such that, $p_{\Pi}^{(m,n)} = p_{\Pi}^{(m,0)} \rtimes p_{\Pi}^{(n,0)}$. Intuitively, this operator swaps the contents of the *comap* part and joins it with the contents from the *map* part. Thus, it suffices to define $p_{\Pi}^{(m,0)}$ and $\mathcal{D}_{\Pi}(m,0)$, to recover all values of p_{Π}^{γ} and $\mathcal{D}_{\Pi}(\gamma)$. In particular

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

where $\mathcal{D}_{\Pi}(m,0) = ((0,n_A),(m_B,0))$ and $\mathcal{D}_{\Pi}(n,0) = ((0,m_A),(n_B,0))$.

We sum up in Figure 3 the values of $\mathcal{D}_{\Pi}(m,0)$.

[1] File `Param_Type.v`

[2] File `Param_Forall.v`

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

Fig. 3. Minimal dependencies for dependent and nondependent product at class $(m,0)$

3.5 The case of non-dependent products

The minimal dependency analysis gives importance to the difference between dependent and non-dependent products. It turns out that in the case of a non-dependent product, it is possible to create a parametricity witness $p^{\gamma}_{\to}$ with less information on the domain A of the arrow type $A \to B$, as well as a function $\mathcal{D}_{\to}(\gamma)$. As a result, using a proof on dependent products on an arrow type would suboptimal, even though it is well-typed which justifies the creation of a special case for the arrow. The dependencies can be found in Figure 3, where we highlight the differences with the general dependent product. We provide the witnesses $p^{\gamma}_{\to}$ in the supplementary material.[3]

4 PARAMETRICITY TRANSLATIONS AS LOGICAL PROGRAMS

In this section we introduce our parametricity translation. Concrete implementations in a functional programming language – such as the coq-param plugin for the Coq proof assistant [Keller and Lasson 2012] implemented in OCaml, or the MetaCoq parametricity plugin [Sozeau et al. 2020], implemented in Coq – however have to manage explicitly the naming bureaucracy left implicit behind the "prime" notation convention.

Because we target an implementation in a logic programming language, we can adopt a sequent style presentation of our translation, which will take care of the "prime" notation in a self-contained way. The overhead between our presentation and the implementation will thus be minimal.

Since we relate our translation to raw parametricity [Bernardy et al. 2012; Bernardy and Lasson 2011; Keller and Lasson 2012] and univalent parametricity [Tabareau et al. 2021], we rephrase them in a sequent style as well, which gradually paves the way to our translation.

4.1 Raw parametricity sequents

We introduce *parametricity contexts*, under the form of a list of triples packaging pairs of related variables with a witness that they are related:

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

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

$$Var(\varepsilon) = \varnothing \qquad Var(\Xi,\ x \sim x' \because x_R) = Var(\Xi) \cup \{x, x', x_R\}$$

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

A *parametricity judgment* relates a parametricity context Ξ with three terms t_1, t_2, t_3. Denoted

$$\Xi \vdash t_1 \sim t_2 \because t_3$$

[3] File `Param_Arrow.v`

it is defined by induction on the syntax of t_1 by the rules given on Figure 4.

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

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

$$\frac{\Xi, x \sim x' \mathrel{\because} x_R \vdash M \sim M' \mathrel{\because} M_R}{\Xi \vdash \lambda x : A.\, M \sim \lambda x' : A'.\, M' \mathrel{\because} \lambda x\, x'\, x_R.\, M_R}\ (\textsc{ParamLam})$$

$$\frac{\Xi \vdash M \sim M' \mathrel{\because} M_R \quad \Xi \vdash N \sim N' \mathrel{\because} N_R}{\Xi \vdash M\ N \sim M'\ N' \mathrel{\because} M_R\ N\ N'\ N_R}\ (\textsc{ParamApp})$$

Fig. 4. Relational-style binary parametricity translation

Lemma 4.1 (Functionality). *The relation associating a term t with pairs (t', t_R) such that $\Xi \vdash t \sim t' \mathrel{\because} t_R$ with Ξ a well-formed parametricity context is* functional*: for any term t and any well-formed Ξ:*

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

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

Definition 4.2 (Admissible parametricity contexts). 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, for any variables x, x', x_R such that $\Xi(x) = (x', x_R)$, and for any terms A' and A_R:

$$\Xi \vdash \Gamma(x) \sim A' \mathrel{\because} A_R \implies \Gamma(x') = A' \wedge \Gamma(x_R) \equiv A_R\ x\ x'$$

We can now state and prove an abstraction theorem:

Theorem 4.3 (Abstraction).

$$\frac{\Gamma \vdash M : A \quad \Xi \vdash M \sim M' \mathrel{\because} M_R \quad \Xi \vdash A \sim A' \mathrel{\because} A_R \quad \Gamma \subset \Delta \triangleright \Xi}{\Delta \vdash M' : A' \quad \textit{and} \quad \Delta \vdash M_R : A_R\ M\ M'}$$

Proof. By induction on the derivation $\Xi \vdash M \sim M' \mathrel{\because} M_R$. □

4.2 Univalent parametricity triples

Before we describe our stratified version of parametricity, we describe a reformulation of univalent parametricity [Tabareau et al. 2021], but in a deductive style, rather than in a functional style. We also change the original definition of the univalent parmetricity record with our equivalent representation. We present this variant in Figure 5, where the statements are of the form

$$(\Gamma, \Xi) \vdash M \sim M' \mathrel{\because} M_R$$

$$\frac{}{(\Gamma, \Xi) \vdash \square_i \sim \square_i \mathrel{\because} p_{\square_i}^{\top,\top}} \text{(UParamSort)}$$

$$\frac{(\Gamma, \Xi) \vdash A \sim A' \mathrel{\because} A_R \qquad (\Gamma', \Xi') \vdash B \sim B' \mathrel{\because} B_R \qquad \Gamma' = \Gamma, a : A, a' : A', a_R : \mathsf{rel}(A_R)\ a\ a' \qquad \Xi' = \Xi, a \sim a' \mathrel{\because} a_R}{(\Gamma, \Xi) \vdash \Pi a : A.\, B \sim \Pi a' : A'.\, B' \mathrel{\because} p_\Pi^\top\ A_R\ B_R} \text{(UParamPi)}$$

$$\frac{(\Gamma, \Xi) \vdash f \sim f' \mathrel{\because} f_R \qquad (\Gamma, \Xi) \vdash a \sim a' \mathrel{\because} a_R}{(\Gamma, \Xi) \vdash f\ a \sim f'\ a' \mathrel{\because} f_R\ a\ a'\ a_R} \text{(UParamApp)}$$

$$\frac{(\Gamma, \Xi) \vdash A \sim A' \mathrel{\because} A_R \qquad (\Gamma', \Xi') \vdash b \sim b' \mathrel{\because} b_R \qquad \Gamma' = \Gamma, a : A, a' : A', a_R : \mathsf{rel}(A_R)\ a\ a' \qquad \Xi' = \Xi, a \sim a' \mathrel{\because} a_R}{(\Gamma, \Xi) \vdash \lambda a : A.\, b \sim \lambda a' : A'.\, b' \mathrel{\because} \lambda a\, a'\, a_R.\, b_R} \text{(UParamLam)}$$

$$\frac{\vdash (\Gamma, \Xi) \qquad x \sim x' \mathrel{\because} x_R \in \Xi}{(\Gamma, \Xi) \vdash x \sim x' \mathrel{\because} x_R} \text{(UParamVar)}$$

Fig. 5. Univalent parametricity rules

where Γ is a CC_ω typing context, Ξ a parametricity context and M, M', and M_R are terms of CC_ω.

We can now state the abstraction theorem for univalent parametricity, rephrased with our notations and definitions.

Theorem 4.4 (Univalent abstraction).

$$\frac{\Gamma \vdash M : A \qquad (\Gamma, \Xi) \vdash M \sim M' \mathrel{\because} M_R \qquad (\Gamma, \Xi) \vdash A \sim A' \mathrel{\because} A_R \qquad \Gamma \subset \Delta \rhd \Xi}{\Delta \vdash M' : A' \qquad \wedge \qquad (\Gamma, \Xi) \vdash M_R : \mathit{rel}(A_R)\ M\ M'}$$

where we recall $\mathit{rel}(A_R)$ is the first projection on A_R, of type

$$\boxdot^\top\ A\ A' \equiv \left(\Sigma R : A \to A' \to \square.\, \mathtt{IsUMap}(R) \times \mathtt{IsUMap}(R^{-1})\right)$$

Note that $\mathsf{rel}(A_R)$ is well defined because

$$\Gamma \vdash A : \square_i \implies \Delta \vdash A_R : \mathsf{rel}(p_{\square_i}^{\top,\top})\ A\ A' \equiv \boxdot^\top\ A\ A'$$

4.3 Annotated type theory

To give a syntactic status to parametricity classes, we define a variant CC_ω^+ of CC_ω where each universe is annotated with a parametricity class, representing the level α of the $\boxdot^\alpha$ relation used as the underlying relation R when translating this universe. For instance, the parametricity witness $p_\square^{(3,1),(0,1)}$ of $\square^{(3,1)}$ at class $(0,1)$ will be an instance of $\boxdot^{(0,1)}\ \square\ \square$ where $\mathsf{rel}(p_\square^{(3,1),(0,1)})]$ is $\boxdot^{(3,1)}$.

$$t, u, A, B \in \mathcal{T}_{CC_\omega^+} ::= \square_i^\alpha \mid \Pi x : A.\, B \mid \lambda x : A.\, t \mid x \mid t\ u$$
$$\alpha \in \mathcal{A} = \{0, 1, 2a, 2b, 3, 4\}^2 \qquad i \in \mathbb{N}$$

The meaning of these annotations is best understood in the context of traversal of a dependent product $\Pi x : A.\, B$ in a parametricity translation. Indeed, before translating B, three terms representing the bound variable x, its translation x', and the parametricity witness x_R will be 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'. In our record-based presentation, the projection $\mathrm{rel}(\cdot)$ is made implicit, so that A_R c an also refer to the relation and not the sigma type. If A is a universe $\square^\alpha$, then this relation is $\boxdot^\alpha$.

Typing terms requires the definition of a *subtyping* relation $\preccurlyeq$ defined by the rules in Figure 6. The typing rules are available in Figure 7 and follow standard presentations [Aspinall and Compagnoni 2001]. The $\equiv$ relation in the ($\textsc{Conv}^+$) rule is the *conversion* relation, defined as the closure of α-equivalence and β-reduction on this variant of λ-calculus.

$$\frac{\alpha \geq \beta}{\Gamma \vdash \square_i^\alpha \preccurlyeq \square_i^\beta}\ (\textsc{SubSort}) \qquad \frac{\Gamma \vdash \Pi x : A.\, B : \square_i \qquad \Gamma \vdash A' \preccurlyeq A \qquad \Gamma, x : A' \vdash B \preccurlyeq B'}{\Gamma \vdash \Pi x : A.\, B \preccurlyeq \Pi x : A'.\, B'}\ (\textsc{SubPi})$$

$$\frac{\Gamma, x : A \vdash t \preccurlyeq t'}{\Gamma \vdash \lambda x : A.\, t \preccurlyeq \lambda x : A.\, t'}\ (\textsc{SubLam}) \qquad \frac{\Gamma \vdash t'\ u : K \qquad \Gamma \vdash t \preccurlyeq t'}{\Gamma \vdash t\ u \preccurlyeq t'\ u}\ (\textsc{SubApp})$$

$$\frac{\Gamma \vdash A : K \qquad \Gamma \vdash B : K \qquad A \equiv B}{\Gamma \vdash A \preccurlyeq B}\ (\textsc{SubConv}) \qquad K ::= \square_i \mid \Pi x : A.\, K$$

Fig. 6. Subtyping rules for CC^+_ω

We first show that our extension is conservative over CC_ω, by defining an erasure function for terms $|\cdot|^- : \mathcal{T}_{CC^+_\omega} \to \mathcal{T}_{CC_\omega}$ and for contexts, defined in Figure 8.

We show that the erasure of subtyping is convertibility in CC_ω:

Lemma 4.5 (Subtyping erasure).

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

Proof. By induction on the derivation. □

$$\frac{(\alpha, \beta) \in \mathcal{D}_\square}{\Gamma \vdash \square_i^\alpha : \square_{i+1}^\beta}\ (\textsc{Sort}^+) \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ (\textsc{Var}^+)$$

$$\frac{\Gamma \vdash A : \square_i^\alpha \qquad \Gamma \vdash B : \square_j^\beta \qquad k = \max(i, j) \qquad \mathcal{D}_\to(\gamma) = (\alpha, \beta)}{\Gamma \vdash A \to B : \square_k^\gamma}\ (\textsc{Arrow}^+)$$

$$\frac{\Gamma \vdash A : \square_i^\alpha \qquad \Gamma, x : A \vdash B : \square_j^\beta \qquad k = \max(i, j) \qquad \mathcal{D}_\Pi(\gamma) = (\alpha, \beta)}{\Gamma \vdash \Pi x : A.\, B : \square_k^\gamma}\ (\textsc{Pi}^+)$$

$$\frac{\Gamma \vdash A : \square_i^\alpha \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.\, t : \Pi x : A.\, B}\ (\textsc{Lam}^+) \qquad \frac{\Gamma \vdash t : \Pi x : A.\, B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B[x := u]}\ (\textsc{App}^+)$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \preccurlyeq B}{\Gamma \vdash t : B}\ (\textsc{Conv}^+)$$

Fig. 7. Typing rules for CC^+_ω

$$\begin{aligned}
\left|\Box_i^C\right|^- &:= \Box_i & |\varepsilon|^- &:= \varepsilon \\
|\Pi x : A.\, B|^- &:= \Pi x : |A|^- .\, |B|^- & |\Gamma, x : A|^- &:= \Gamma, x : |A|^- \\
|\lambda x : A.\, B|^- &:= \lambda x : |A|^- .\, |B|^- \\
|T\ U|^- &:= |T|^-\ |U|^- \\
|x|^- &:= x
\end{aligned}$$

Fig. 8. Erasure function from CC^+_ω to CC_ω

Finally we show that our extension is conservative

Theorem 4.6 (Annotation erasure).

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

Proof. By induction on the derivation. □

4.4 Stratified parametricity relation

We update the parametricity relation to a quadruplet $t \ @\ T \sim t' \because t_R$ where t, T, t' and t_R are terms in CC^+_ω . We also update the parametricity context Ξ to a list of these quadruplets, and the parametricity judgment now states that from the updated context, 4 terms are related in the updated relation. Hence, stratified parametricity judgments are of the form $(\Gamma, \Xi) \vdash t \ @\ T \sim t' \because t_R$. Figure 9 displays all the rules for the stratified parameticity translation in CC_ω.

The second term of the relation T guides the translation through the various annotations on $\Box$. If a specific translation is not available, we may weaken another one in order to obtain an element of $\boxdot^\alpha$. We start with describing the weakening function.

Weakening. From the lattice and the subtyping relation for CC^+_ω , we can define a weakening function for the parametricity witness.

First, we define a function $\downarrow^p_q$ to weaken the map records from a level p to a lower level q in the hierarchy (*i.e.*, the function is defined for $p \geq q$), by forgetting the right number of fields. We can then define a function $\Downarrow^\alpha_\beta$ which composes twice the previous one on each side of the parametricity record:

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

We can build the weakening function on any parametricity witness by extending this last function to all annotated types, *i.e.*, $\Downarrow^T_U$ such that $T \preccurlyeq U$, and we describe the full definition of $\Downarrow^T_U$ in Figure 10.

Erasure. We show that our stratified parametricity relation entails raw parametricity after all annotations are erased.

Theorem 4.7 (Erasure of stratification).

$$\forall t, A, t', t_R \in \mathcal{T}_{CC^+_\omega}, \quad (\Gamma, \Xi) \vdash t \ @\ A \sim t' \because t_R \implies \Xi \vdash |t|^- \sim \left|t'\right|^- \because rel^*(t_R)$$

where

$$\begin{aligned}
rel^*(MN) &:= rel^*(M)\ rel^*(N) \\
rel^*(\lambda x.t) &:= \lambda x.rel^*(t) \\
rel^*(t) &:= rel(t)
\end{aligned}$$

$$\frac{(\alpha,\beta) \in \mathcal{D}_{\square}}{(\Gamma,\Xi) \vdash \square_i^{\alpha} \mathbin{@} \square_{i+1}^{\beta} \sim \square_i^{\alpha} \mathrel{\because} p_{\square_i}^{\alpha,\beta}} \,(\textsc{ParamSort}^+)$$

$$\frac{(\alpha,\beta) = \mathcal{D}_{\to}(\gamma) \qquad k = \max(i,j) \qquad (\Gamma,\Xi) \vdash A \mathbin{@} \square_i^{\alpha} \sim A' \mathrel{\because} A_R \qquad (\Gamma,\Xi) \vdash B \mathbin{@} \square_j^{\beta} \sim B' \mathrel{\because} B_R}{(\Gamma,\Xi) \vdash A \to B \mathbin{@} \square_k^{\gamma} \sim A' \to B' \mathrel{\because} p_{\to}^{\gamma}\, A_R\, B_R} \,(\textsc{ParamArrow}^+)$$

$$\frac{\begin{array}{c}(\alpha,\beta) = \mathcal{D}_{\Pi}(\gamma) \qquad k = \max(i,j) \\ (\Gamma,\Xi) \vdash A \mathbin{@} \square_i^{\alpha} \sim A' \mathrel{\because} A_R \qquad (\Gamma',\Xi') \vdash B \mathbin{@} \square_j^{\beta} \sim B' \mathrel{\because} B_R \\ \Gamma' = \Gamma, a : A, a' : A', a_R : A_R\ a\ a' \qquad \Xi' = \Xi, a \mathbin{@} A \sim a' \mathrel{\because} a_R\end{array}}{(\Gamma,\Xi) \vdash \Pi a : A.\, B \mathbin{@} \square_k^{\gamma} \sim \Pi a' : A'.\, B' \mathrel{\because} p_{\Pi}^{\gamma}\, A_R\, B_R} \,(\textsc{ParamPi}^+)$$

$$\frac{(\Gamma,\Xi) \vdash f \mathbin{@} \Pi a : A.\, B \sim f' \mathrel{\because} f_R \qquad (\Gamma,\Xi) \vdash a \mathbin{@} A \sim a' \mathrel{\because} a_R}{(\Gamma,\Xi) \vdash f\ a \mathbin{@} B \sim f'\ a' \mathrel{\because} f_R\ a\ a'\ a_R} \,(\textsc{ParamApp}^+)$$

$$\frac{\begin{array}{c}(\Gamma,\Xi) \vdash A \mathbin{@} \square_i^{\alpha} \sim A' \mathrel{\because} A_R \qquad (\Gamma',\Xi') \vdash b \mathbin{@} B \sim b' \mathrel{\because} b_R \\ \Gamma' = \Gamma, a : A, a' : A', a_R : A_R\ a\ a' \qquad \Xi' = \Xi, a \mathbin{@} A \sim a' \mathrel{\because} a_R\end{array}}{(\Gamma,\Xi) \vdash \lambda a : A.\, b \mathbin{@} \Pi a : A.\, B \sim \lambda a' : A'.\, b' \mathrel{\because} \lambda a\, a'\, a_R.\, b_R} \,(\textsc{ParamLam}^+)$$

$$\frac{\vdash (\Gamma,\Xi) \qquad x \mathbin{@} T \sim x' \mathrel{\because} x_R \in \Xi}{(\Gamma,\Xi) \vdash x \mathbin{@} T \sim x' \mathrel{\because} x_R} \,(\textsc{ParamVar}^+)$$

$$\frac{(\Gamma,\Xi) \vdash t \mathbin{@} T \sim t' \mathrel{\because} t_R \qquad \Gamma \vdash T \preccurlyeq T'}{(\Gamma,\Xi) \vdash t \mathbin{@} T' \sim t' \mathrel{\because} \Downarrow_{T'}^{T} t_R} \,(\textsc{ParamSub}^+)$$

Fig. 9. Stratified parametricity rules

$$\begin{aligned}
\Big\Downarrow_{\square_{i'}^{C'}}^{\square_i^{C}} t_R &:= \Downarrow_{C'}^{C} t_R \\
\Big\Downarrow_{\Pi x:A'.\,B'}^{\Pi x:A.\,B} t_R &:= \lambda x\, x'\, x_R.\, \Big\Downarrow_{B'}^{B} \left(t_R\ x\ x'\ \left(\Big\Downarrow_{A}^{A'} x_R\right)\right) \\
\Big\Downarrow_{T'\ U'}^{T\ U} t_R &:= \Big\Downarrow_{T'}^{T} U\ U'\ t_R \\
\Big\Downarrow_{\lambda x:A'.\,B'}^{\lambda x:A.\,B} T\ T'\ t_R &:= \Big\Downarrow_{B'[x:=T']}^{B[x:=T]} t_R \\
\Big\Downarrow_{T'}^{T} t_R &:= t_R
\end{aligned}$$

Fig. 10. Weakening function

$$\begin{aligned}
|\square_i|^+ &:= \square_i^{\top} \\
|\Pi x : A.\, B|^+ &:= \Pi x : |A|^+ .\, |B|^+ \\
|\lambda x : A.\, B|^+ &:= \lambda x : |A|^+ .\, |B|^+ \\
|T\ U|^+ &:= |T|^+\ |U|^+ \\
|x|^+ &:= x \\
\\
|\varepsilon|^+ &:= \varepsilon \\
|\Gamma, x : A|^+ &:= \Gamma, x : |A|^+
\end{aligned}$$

Fig. 11. Maximal annotation function from CC_ω to CC_ω^+

Proof. By induction on the derivation. □

Recovering univalent parmetricity. We show we can recover univalent parametricity by defining a maximal annotation function for terms $|\cdot|^+ : \mathcal{T}_{CC_\omega} \to \mathcal{T}_{CC^+_\omega}$ and for contexts, defined in Figure 11.

Indeed, we have the following theorem.

Theorem 4.8 (Maximally annotated stratified parametricity).

$$\forall t, A, t', t_R \in \mathcal{T}_{CC_\omega}, \quad (\Gamma, \Xi) \vdash |t|^+ \ @ \ |A|^+ \sim \left|t'\right|^+ \because |t_R|^+ \iff \Xi \vdash_u t \sim t' \because t_R$$

Proof. By induction on the derivation. □

Abstraction theorem for stratified parametricity. Finally, we state our abstraction theorem:

Theorem 4.9 (Stratified Abstraction).

$$\frac{\Gamma \vdash M : A \qquad (\Gamma, \Xi) \vdash M \ @ \ A \sim M' \because M_R \qquad (\Gamma, \Xi) \vdash A \ @ \ \square_i^\alpha \sim A' \because A_R \qquad \Gamma \subset \Delta \rhd \Xi}{\Delta \vdash M' : A' \qquad \wedge \qquad \Delta \vdash M_R : \mathit{rel}(A_R)\ M\ M'}$$

Proof. By induction on the derivation. □

By applying the rule with $\Gamma \vdash A : \square^\alpha$ we get:

$$\frac{\Gamma \vdash A : \square^\alpha \qquad (\Gamma, \Xi) \vdash A \ @ \ \square^\alpha \sim A' \because A_R \qquad (\Gamma, \Xi) \vdash \square^\alpha \ @ \ \square^\beta \sim \square^\alpha \because p_\square^{\alpha,\beta} \qquad \Gamma \subset \Delta \rhd \Xi}{\Delta \vdash A' : \square^\alpha \qquad \wedge \qquad \Delta \vdash A_R : \mathsf{rel}(p_\square^{\alpha,\beta})\ A\ A'}$$

By definition $\mathsf{rel}(p_\square^{\alpha,\beta}) = \boxdot^\alpha$, hence $\Gamma \vdash A_R : \boxdot^\alpha\ A\ A'$, as expected by the initial type annotation. Therefore, by applying the rule with $\Gamma \vdash \square^\alpha : \square^\beta$ we get $\Gamma \vdash p_\square^{\alpha,\beta} : \boxdot^\beta\ \square^\alpha\ \square^\alpha$ as expected, provided that $(\alpha, \beta) \in \mathcal{D}_\square$.

4.5 Handling constants on top of CC^+_ω

For concrete applications, we must take constants into our calculus and translations. The case of constants is very similar to the case of variables, except that they are not stored in the context Γ, and they can be annotated by several different types.

For example, the constant `list`, such that `list` A represents the type of lists of elements of can be annotated with the type $\square^\alpha \to \square^\alpha$ for any $\alpha \in \mathcal{A}$.

We pose a set of constants $\mathcal{C}$, and its possible annotated types $T_\cdot : \mathcal{C} \to \mathfrak{P}\left(\mathcal{T}_{CC^+_\omega}\right)$, with the property that all possible annotated types of a constant have the same erasure in CC_ω, *i.e.*, $\forall c, \forall A, \forall B,\ A, B \in T_c \Rightarrow |A|^- = |B|^-$. For example, $T_{\texttt{List}} = \{\square^\alpha \to \square^\alpha \mid \alpha \in \mathcal{A}\}$.

Additionally, we provide translations $\mathcal{D}_c(A)$ for each possible annotation T_c of each constant c. For example $\mathcal{D}_{\texttt{List}}(\square^{(1,0)} \to \square^{(1,0)})$ is well defined and equal to the translation

$$\left(\texttt{list}, \qquad \lambda A\, A'\, A_R.\ (\texttt{List.All2}\ A_R,\ \texttt{List.map}\ (\texttt{map}(A_R))),\right)$$

where `List.All2` A_R relates lists that are pointwise related by A_R, `List.map` is the standard map function on lists and $\texttt{map}(A_R) : A \to A'$ extracts the *map* projection of the record A_R of type $\boxdot^{(1,0)}\ A\ A' \equiv \Sigma R.\, A \to A'$.

We describe in Figure 12 the additional rules for constants in CC^+_ω and the stratified parametricity translation. Note that the wrong choice of annotation may lead to the absence of translation for a given term containing constants. For example, we cannot pick the annotated type $\square^{(1,0)} \to \square^{(1,0)}$ in order to translate `list` $@\ \square^{(2,0)}$.

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

Fig. 12. Additional constant rules for CC^+_ω and stratified parametricity

5 IMPLEMENTATION IN COQ-ELPI

We chose the Coq-Elpi meta-language to implement a prototype for Trocq. In this section, we first introduce Coq-Elpi (§ 5.1), then we show how we setup the plugin and automatically generate Coq definitions (§ 5.2), before explaining how we encode and solve the inference problem (§ 5.3). Finally, we conclude with the advantages of the logic programming paradigm in our formulation of parametricity translation (§ 5.4).

5.1 Introducing Coq-Elpi

Coq-Elpi is one of the major meta-languages available for Coq, amongst MetaCoq [Sozeau et al. 2020], Ltac 2 [The Coq Development Team 2022], Mtac [Kaiser et al. 2018], etc. At its heart lies the Elpi language, an implementation of λProlog, a functional logic programming language well suited to handle abstract syntax trees containing binders and unification variables [Dunchev et al. 2015]. Indeed, as we shall see later in this section, Coq terms are encoded in *Higher Order Abstract Syntax* (HOAS) in Coq-Elpi, and Elpi features the introduction of both locally bound variables, called *universal constants*, and local information about them through the logical implication of Prolog, allowing to traverse terms with binders having full typing information on the bound variables, and forging terms without ever having to manage De Bruijn indices or worry about well-formedness of a subterm.

Coq-Elpi is an extension around Elpi to turn it into a full-fledged plugin for Coq. It features built-in predicates to interact with the proof assistant, *e.g.*, searching for existing definitions, calling the typechecker or elaborator, or creating new definitions. It also allows declaring commands taking arbitrary arguments, and tactics receiving the proof state as input and being able to make proof steps.

Another feature, particularly interesting in our context, is the possibility to create databases represented as banks of predicate instances. The user can fill a database by calling Elpi code or commands designed to perform updates on this database. The content of a database can be accessed during execution of commands or tactics linked to this database, thus enabling a plugin architecture that is relevant for our use case: creating commands to store information given by the Coq user, and tactics to perform a specific task in light of this information.

5.2 Setting up the plugin

Before actually implementing the core translation and inference algorithms at the meta-level, we need to make various definitions in Coq to represent concepts used in the parametricity rules.

First, we represent parametricity classes with two values in a `map_class` inductive type with values ranging from `map0` to `map4`, representing the values in $\mathcal{A}$. Annotated types can therefore be represented as standard Coq types where all occurrences of `Type` have been replaced with an application `PType` $\mathtt{m}_1$ $\mathtt{m}_2$ where the arguments are phantom values whose single purpose is that we can match them syntactically, *i.e.*, this application is definitionally equal to the previous term, and the annotation can be erased in a step of δ-reduction. Annotated terms being valid Coq terms will allow calling the Coq typechecker at various places during the translation.

Second, all the families of proofs indexed with parametricity classes (*e.g.*, p_Π) are entirely generated when the plugin is compiled. Proofs for each possible field of the record are written by hand, and an Elpi program makes all possible combinations M_i of these fields (see Definition 3.7) in order to make proofs for all parametricity classes.

5.3 Encoding and solving the parametricity class inference problem

The stratified parametricity translation does not assign the various parametricity classes found in the annotated types, but only constrains them, almost always leaving multiple valid solutions at the end of the term traversal. The most interesting solution is the one minimizing all the parametricity classes, as it will also minimize the amount of information required from the user, and in particular, determine whether axioms are really needed to translate a term.

5.3.1 A finite domain constraint solving problem. An encoding of this problem can be made where every unknown parametricity class is a variable, with an initial domain ranging from $(0, 0)$ to $(4, 4)$. The constraints found in the inference rules reduce this domain, leaving only the valid solutions. In the end of the translation, the valid solution is an association from each variable to the lowest remaining class in its domain. This very much resembles finite domain constraint solving problems, and as such could be solved elegantly by implementing an *ad hoc* constraint solver in the style of constraint logic programming [Jaffar and Lassez 1987], which is idiomatic in Prolog-based languages like Elpi. One of the most well-known methods to express complex algorithms involving constraint generation and handling with reduction rules is the *Constraint Handling Rules* (CHR) language [Frühwirth and Raiser 2011], 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. The full constraint solver can be obtained by combining these rules with a search procedure, testing values in the domains remaining after propagation. In our case, the reduction rules are there to simplify the complex constraints and reduce them to ordering constraints on the variable classes. Indeed, once a class C is known, a constraint $(C, C_R) \in \mathcal{D}_\square$ can be turned into $C_R \geq (4, 4)$ or no constraint at all, and values $\mathcal{D}_\Pi(C)$ or $\mathcal{D}_\to(C)$ can be computed and turned into new constraints for subterms.

However, the elegance of such a solution comes at the cost of trackability of the control flow in the reduction process. In constraint solvers, when constraints are declared, after an initial propagation phase, they actually remain in the constraint store in an asleep state, watching the variables they bound together, to be awaken every time one of their domains is updated. This behaviour is necessary to ensure global consistency between the remaining domains of the variables and all the constraints declared in the store, but it makes control flow very complex as it is difficult to know when propagation will happen just by reading the code, thus making debugging phases harder.

5.3.2 Accumulation of constraints and graph reduction. Instead, we present an inference algorithm in a more pragmatic style, first accumulating constraints in a global constraint graph, then reducing it and instantiating the variables after the translation. In this constraint graph, 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 13. 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 $(3, 2\mathrm{b}) \geq X_6$.

Because of the fact that complex constraints can only be reduced to ordering constraints once their first variable is instantiated, not all the ordering constraints are known in the fully accumulated

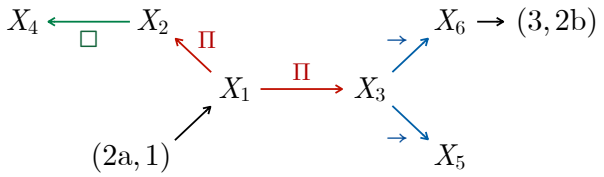

Fig. 13. Example of constraint graph

constraint graph at the end of the translation. Yet, all the ordering constraints linked to a variable must be known in order to choose the minimal class for this variable. As a result, we need to process the variables in a precise order, *i.e.*, reduce the graph at each instantiation by removing the node of the freshly instantiated variable and computing new constraints on the other variables, so as to know all the ordering constraints applying to a variable at the time of its instantiation. To that end, we can define an order of dependency:

- $(\alpha, \beta) \in \mathcal{D}_\square$ forces α to be instantiated before β;
- $\mathcal{D}_\Pi(\gamma) = (\alpha, \beta)$ or $\mathcal{D}_\to(\gamma) = (\alpha, \beta)$ force γ to be instantiated before α and β;
- a $\alpha \geq \beta$ constraint forces β to be instantiated before α, as the lower bound of the domain is the one that matters, and the instantiation of β can update the lower bound for α.

The direction of the edges in the constraint graph is already consistent with this ordering, meaning that we can build a precedence graph from the constraint graph, where there is an edge from X to Y if there is a constraint of any type from X to Y in the constraint graph. The final instantiation order can then be obtained by performing a *topological sort* on this precedence graph.

When instantiating a variable, all the edges pointing to this variable (*i.e.*, all the lower nodes) form a set of constant classes. The minimal acceptable class is the least upper bound of all these constant classes. Once it is known, all the constraints represented by the edges starting from the current variable can be removed from the graph, simplified as ordering constraints which are re-added to the graph, so that the next variable is always ready to be instantiated.

5.4 Logic programming to the rescue

The logic programming paradigm on which Elpi is based, is ideal to implement algorithms expressed as inference rules, as each inference rule can be associated to an instance of a predicate. In our case, the translation comes in the form of a `param` predicate of arity 4, where `param X T X' XR` represents the inference of $x \mathbin{@} T \sim x' \mathbin{\because} x_R$, where x and T are input values (initially, the source goal and $\square^{(0,1)}$), and the translated term x' and the witness x_R are outputs. Several instances of the predicate are defined to cover the various inference rules present in the theoretical presentation.

As an example, let us inspect the instance of the `param` predicate in charge of translating dependent products in our implementation, and compare it with the corresponding case in the inference rules, *i.e.*, rule ParamPi$^+$ in Figure 9. Note that for more readability, we give the code in a compressed but equivalent version, without lines related to logs, pretty-printing, and fresh universe instance generation, *i.e.*, the parts that would not bring much information if included in the article. First, let us see the head of the predicate:

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

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

Another difference between the theoretical presentation and the implementation is the necessity to track binders correctly, *i.e.*, B is not a valid term on its own, it must carry a context. Thankfully, in Coq-Elpi, the HOAS encoding of Coq terms gives the variable `B` the type `term -> term`, which means we never have to worry about a term with bound variables being ill-typed. In addition, the context is implicit thanks to the functional logic programming paradigm, allowing to locally add a new typed variable to the context with a `pi` operator and the implication (`=>`). The new variables cannot escape their scope: as we can see, the return values of the recursive call on `B` are also meta-functions `B'` and `BR`, abstracting the previously locally introduced variables that might occur in their content.

The last step is to build the output proof $p^C_\Pi\ A_R\ B_R$, which is a bit more verbose in the code than on paper. Indeed, as the axioms that might be involved in some proofs are not included globally, which would make inference meaningless, they are used as an additional argument in the proofs that require them. Therefore, we need to check the output parametricity class C to know if it requires the addition of an axiom to the list of arguments (in the case of the dependent product, function extensionality):

```
Args =
  [ A, A', AR, fun N A B, fun _ A' B',
    fun N A a\ fun _ A' a'\
    fun _ (app [pglobal (const {param.db.r CA}) _, A, A', AR, a, a']) aR\
      BR a a' aR ],
util.if-suspend C (param-class.requires-axiom C)
  (Args' = [pglobal (const {param.db.funext}) _|Args])
  (Args' = Args),
Prod' = prod _ A' B',
ProdR = app [pglobal (const {param.db.param-forall C}) _|Args'].
```

If the user has not added any axiom at the time of calling `param.db.funext`, the code rightfully fails because the translation is impossible. Otherwise, the body of the predicate terminates with a success.

6 APPLICATIONS

In this section we will show how Trocq covers a series of examples. First we sketch the description of a class of statments that we conjecture can be translated without univalence. We illustrate this with Example 1.1, which performs a transfer by isomorphism, and Example 1.2 which transfers a result to a subtype.

The supplementary material also contains an example of transfer from $\mathbb{Z}$ to a quotient $\mathbb{Z}/p\mathbb{Z}$[4], as well as examples in which we recover the behaviour of setoid rewriting[5] and generalized rewriting[6] at the price of a manual setup to perform pattern recognition.

6.1 A class of univalence-free transferable statements

Note that none of the above examples require the use of univalence, as the statements we transfer fall in a fragment of formulas $\mathcal{F}$ defined as follows.

Definition 6.1 (Univalence-free formulas $\mathcal{F}$.). We define the grammar of these formulæ using the non terminal symbols $\mathcal{F}_k$ for $k \in \{1, 2a, 2b, 4\}$

$$\begin{aligned}
\mathcal{F}_1 &:= \square \mid \Pi(x : \mathcal{F}_{2a}).\mathcal{F}_1 \mid \mathcal{F}_1 \to \mathcal{F}_1 \mid x\vec{M} \mid \mathcal{F}_{2a} \mid \mathcal{F}_{2b} \\
\mathcal{F}_{2a} &:= \square \mid \Pi(x : \mathcal{F}_4).\mathcal{F}_{2a} \mid \mathcal{F}_{2b} \to \mathcal{F}_{2a} \mid x\vec{M} \mid \mathcal{F}_4 \\
\mathcal{F}_{2b} &:= \Pi(x : \mathcal{F}_{2a}).\mathcal{F}_{2b} \mid \mathcal{F}_{2b}\mathcal{F}_{2b} \mid x\vec{M} \mid \mathcal{F}_4 \\
\mathcal{F}_4 &:= \Pi(x : \mathcal{F}_4).\mathcal{F}_4 \mid x\vec{M}
\end{aligned}$$

Conjecture 6.2 (Univalence-free transfer). *For all A, such that $|A|^{-} \in \mathcal{F}_k$, $\Gamma \vdash A : \square$ and $(\Gamma, \Xi) \vdash A \mathbin{@} \square^{(k,k)} \sim A' \because A_R$, there exists a translation $(\Gamma, \Xi) \vdash A \mathbin{@} \square^{(k,k)} \sim B' \because B_R$ such that B_R does not use the axiom of univalence.*

Additionally, we can add some constants, such as base types like $\mathbb{N}$, and containers (like `option`, `list`, etc) to the syntax of $\mathcal{F}_k$ for all k.

6.2 Example 1.1: transferring induction principles

With notations from the introduction[7], we recall that we need to prove

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

A quick analysis shows that this statement is indeed in $\mathcal{F}_1$.

In addition to the notions already described in the introduction, we need to inform Trocq of three facts:

```
N_R : Param2a3.Rel N ℕ
O_R : rel N_R O_N O_ℕ
S_R : ∀ m n, rel N_R m n → rel N_R (S_N m) (S_ℕ n).
```

The first fact is that there is a split injection from `N` to `ℕ`, the second that the zeros are related, and the third that the sucessors are related.

And this is enough for Trocq to generate and apply the implication:

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

[4] File `int_to_Zp.v`

[5] File `trocq_setoid_rewrite.v`

[6] File `trocq_gen_rewrite.v`

[7] See file `peano_bin_nat.v` in the supplementary material.

6.3 Example 1.2: transferring results to a subtype

We setup an axiomatic context in Appendix 7 so as to state the goal on the type $\mathbb{R}_{\geq 0}$ of positive reals.[8] We prove the relation between this type and its extended $\overline{\mathbb{R}}_{\geq 0}$, their respective binary additions, infinite sums, declare them to Trocq. We can then prove:

```
Lemma Σ_ℝ≥0_add : forall u v : summable, Σ_ℝ≥0 (u + v) = Σ_ℝ≥0 u + Σ_ℝ≥0 v.
Proof. trocq; exact: Σ_ℝ̄≥0_add. Qed.
```

7 RELATED WORK AND PERSPECTIVES

The functionalities of the prototype plugin presented in § 5 can be extended in several directions. It would be particularly fruitful to connect it with tools able to automate the generation of equivalence proofs, such as Pumpkin Pi [Ringer et al. 2021]. Other improvements, *e.g.*, addressing the case of Coq's impredicative sort, involve non-trivial implementation issues, related to Coq's management of universe polymorphism. We now discuss how the current state of this prototype compares with other implemented approaches to proof transfer in interactive theorem proving, listed in chronological order in the summary Table 1. For each such tool, the table indicates whether a given feature is available (✓), not available (✗) or only partially available (🖉 and ➡).

In the context of type theory, the idea that the computational content of type isomorphisms can be used for proof transfer already appears in [Barthe and Pons 2001]. The first implementation report of a tool based on this idea appeared soon after [Magaud 2003]. Implemented in a meta-language and based on proof rewriting, this heuristic translation was producing a candidate proof term from a given proof term, with no formal guarantee, not even that of being well-typed. As mentioned in § 2.1, generalized rewriting [Sozeau 2009], which generalizes setoid rewriting to preorders, is also 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 applications to quasi equivalence relations [Krishnaswami and Dreyer 2013], or to datatype representation change.

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, as illustrated in § 2.1, although this special case is seldom emphasized. The Coq Effective Algebra Library (CoqEAL) [Cohen et al. 2013; Dénès et al. 2012] and the Isabelle/HOL transfer package [Haftmann et al. 2013; Huffman and Kunčar 2013; Lammich 2013; Lammich and Lochbihler 2019], 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 [Zimmermann and Herbelin 2015] 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 trivial QPER in which the zig-zag morphism is a partial surjection from A to B.

The next two columns of the table concern proof transfer in presence of the univalence principle, either axiomatic, in the case of univalent parametricity [Tabareau et al. 2021] or computational, in the case of [Angiuli et al. 2021]. Key ingredients of the univalent parametricity were already present in earlier seemingly unpublished work [Anand and Morrisett 2017], implemented using an outdated ancestor of the MetaCoq library [Sozeau et al. 2020].

Table 1 indicates which tools can transfer along *heterogeneous relations*, as this is a prerequisite to changing type representation, and which ones operate by proving an *internal* implication lemma, as opposed to a monolithic translation of an input proof term. We borrow the terminology used in [Tabareau et al. 2021], in which *anticipation* refers to the need to define a dedicated structure for

[8] Also see file `summable.v`

the signature to be transported. *Binders* can prevent transfer, as well as *dependent types*. The latter are recovered in presence of univalence. The first published publication [Tabareau et al. 2018] on the univalent parametricity translation suggested that the translation does not pull the axiom in when translating terms in the F^{ω} fragment. However, we identified a strictly larger class of terms for which the stratified approach can get rid of it. Finally, the table indicates which approaches can deal with *quasi-equivalence relations* (QER), and with (explicit) subtyping relations.

	[Magaud 2003]	Setoid rewrite [Sozeau 2009]	CoqEAL [Cohen et al. 2013]	Isabelle/HOL Transfer (2013)	[Zimmermann and Herbelin 2015]	[Tabareau et al. 2021]	[Angiuli et al. 2021]	Trackt [Blot et al. 2023]	Trocq (2023)
Heterogeneous relations	✓	✗	✓	✓	✓	✓	✓	✓	✓
Internal	✗	✓	✓	✓	✓	✓	✓	✓	✓
No anticipation	✓	✓	✓	✓	✓	✓	✗	✓	✓
Substitution under ∀	✓	✓	✗	✓	✓	✓	✓	✓	✓
Substitution in dep. types	✓	✗	✗	✗	✗	✓	✓	✗	✓
No univalence for ?	✓	✓	✓	✓	✓	✗	✗	✓	✓
Preorder relations	✗	✓	?	?	?	✗	?	?	🖉
Subrelations	✗	✓	✗	✗	✗	✗	✗	✗	🖉
QERs	✗	🖉	➡	➡	➡	✗	✓	✗	➡
Subtyping relations	✗	✗	➡	➡	➡	✗	✗	➡	➡
System	Coq	Coq	Coq	Isabelle/HOL	Coq	Coq/HoTT	(Cubical) Agda	Coq	Coq or Coq/HoTT

Table 1. Comparison of proof transfer automation devices

In its current state, the Trocq plugin can already address the proof transfer bureaucracy of state-of-the-art formal proofs of computational mathematics, such as the one described in [Allamigeon et al. 2023]. However this framework opens the way to perform unification modulo both generalized rewriting and heterogeneous transfer relations, potentially solving the problem coined by the community as *concept alignment*. We expect that our work, once put in production makes it possible to have the same lemma applicable to a wide variety of different types: isomorphic types, subtypes, and quotient types. A particular pervasive problem we expect to solve is the identification of the canonical natural number with the canonical natural number object in several types, *e.g.*, $\{x : \mathbb{R} \mid \exists n : \mathbb{N}, x = \iota(n)\}$, etc. Another one is the identification of objects constructed in different ways, which happen to be the same on a subset of their arguments, *e.g.*, the ring $\mathbb{Z}/q\mathbb{Z}$, defined for all $q > 0$ and the Galois Field $\mathbb{F}_q$, defined when $q = p^k$, which happen to be canonically isomorphic if and only if q is prime.

REFERENCES

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. https://doi.org/10.1145/3372885.3373824

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.

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. https://doi.org/10.1145/3573105.3575678

Abhishek Anand and Greg Morrisett. 2017. Revisiting Parametricity: Inductives and Uniformity of Propositions. arXiv:1705.01163 [cs.LO]

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

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

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

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.

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. https://doi.org/10.1145/3018610.3018615

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

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. https://doi.org/10.1007/978-3-642-19805-2_8

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. https://doi.org/10.1145/3573105.3575676

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. https://doi.org/10.1145/3018610.3018620

Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2017. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. *FLAP* 4, 10 (2017), 3127–3170. http://collegepublications.co.uk/ifcolog/?00019

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. https://doi.org/10.1007/978-3-319-03545-1_10

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

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. https://doi.org/10.1007/978-3-030-79876-5_37

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.

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. https://doi.org/10.1007/978-3-662-48899-7_32

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

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

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.

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.

Joxan Jaffar and J-L Lassez. 1987. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 111–119.

Jan-Oliver Kaiser, Beta Ziliani, Robbert Krebbers, Yann Régis-Gianas, and Derek Dreyer. 2018. Mtac2: typed tactics for backward reasoning in Coq. *Proc. ACM Program. Lang.* 2, ICFP (2018), 78:1–78:31. https://doi.org/10.1145/3236773

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. https://doi.org/10.4230/LIPIcs.CSL.2012.381

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. https://doi.org/10.4230/LIPIcs.CSL.2013.432

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.

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. https://doi.org/10.1007/s10817-018-9461-9

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.

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. https://doi.org/10.1007/s10817-015-9350-4

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. https://doi.org/10.1145/512644.512669

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

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. https://doi.org/10.1007/978-3-642-04652-0_5

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.

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. https://doi.org/10.1145/3453483.3454033

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

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. https://doi.org/10.1007/s10817-019-09540-0

Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. 2018. Equivalences for free: univalent parametricity for effective transport. *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 1–29.

Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. 2021. The marriage of univalence and parametricity. *Journal of the ACM (JACM)* 68, 1 (2021), 1–44.

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. https://doi.org/10.4230/

LIPIcs.CVIT.2016.23

The Coq Development Team. 2022. *The Coq Proof Assistant.* https://doi.org/10.5281/zenodo.7313584

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

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. https://doi.org/10.1145/3341691

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. https://doi.org/10.1145/99370.99404

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

APPENDIX

```
(* We postulate the bare minimum about non-negative reals *)
Axioms (ℝ≥0 : Type) (0ℝ≥0 : ℝ≥0) (+ℝ≥0 : ℝ≥0 → ℝ≥0 → ℝ≥0).
(* Non-negative extended reals are a trivial extension *)
Inductive ℝ̄≥0 : Type := Fin : ℝ≥0 → ℝ̄≥0 | Inf : ℝ̄≥0.
(* We define the notions of sequences of numbers *)
Definition seqℝ̄≥0 := nat → ℝ̄≥0.
Definition seqℝ≥0 := nat → ℝ≥0.
(* Addition on the extended non-negative reals is definable *)
Definition a +ℝ̄≥0 b : ℝ̄≥0 := match a, b with
  Fin x, Fin y ⇒ Fin (r1 +ℝ≥0 r2) | _, _ ⇒ Inf end.
(* We can derive the addition on sequences *)
Definition u +seqℝ̄≥0 v : seqℝ̄≥0 := fun n ⇒ u n +ℝ̄≥0 v n.

(* We now postulate the unconditional infinite summation
    on extended non-negative reals and its linearity *)
Axiom Σℝ̄≥0 : seqℝ̄≥0 → ℝ̄≥0.
Axiom Σℝ̄≥0_add : forall u v : seqℝ̄≥0, Σℝ̄≥0 (u + v) = Σℝ̄≥0 u + Σℝ̄≥0 v.

(* We define the notion of summable sequence *)
Definition isFin (a : ℝ̄≥0) := match a with Fin _ ⇒ true | _ ⇒ false end.
Definition truncate (a : ℝ̄≥0) := match a with Fin x ⇒ x | _ => 0ℝ≥0 end.
Definition isSummable (u : seqℝ≥0) := isFin (Σℝ̄≥0 (Fin ∘ u)).

(* We define the type of summable sequences *)
Record summable := {to_seq :> seqℝ≥0; _ : isSummable to_seq}.

(* we postulate that summability is preserved by binary addition *)
Axiom summable_add :
  forall u v : summable, isSummable (fun n ⇒ u n +ℝ≥0 v n) = true.
Definition u +summable v : summable := Build_summable _ (Σℝ̄≥0_add u v).

(* Finally, we define infinite sums on summable sequences *)
Definition Σℝ≥0 (u : summable) : ℝ≥0 := truncate (Σℝ̄≥0 (Fin ∘ u)).
```