Trocq: a deeper study of the lattice of relations

Lucie Lahaye, Cyril Cohen

▶ **To cite this version:**

Lucie Lahaye, Cyril Cohen. Trocq: a deeper study of the lattice of relations. Formal Languages and Automata Theory [cs.FL]. 2025. ⟨hal-05342516⟩

HAL Id: hal-05342516

https://inria.hal.science/hal-05342516v1

Submitted on 2 Nov 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons CC BY 4.0 - Attribution - International License

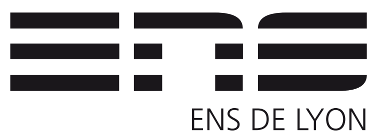
Lucie Lahaye ENS Lyon

M2 Internship report

Trocq: a deeper study of the lattice of relations

Internship referent : **Cyril Cohen**

Research team : CASH, Laboratoire d'Informatique et du Parallélisme (LIP), ENS Lyon



I. Introduction

The discipline of formal verification can be defined broadly as the wide range of techniques useful to obtain confidence in theorical results, often using a computer system. For instance, proofs about mathematical results can be input in a computer and their verification mecanized. This enables getting rid of human errors. For instance, it is thus widely used in the industry to ensure security requirements [1].

During this internship, I studied and worked on *Trocq*, a plugin for *Rocq* [2]. Rocq is a general-purpose proof-assistant. Rocq implements a calculus system – CIC, the Calculus of Inductive Constructions [3, 4] – which provides a powerful theoretical framework for the development of formal proofs. Some famous developments made using Rocq include the proof of the four-color theorem [5, 6], a formalization of the Odd Order theorem [7] (originally proved by Felt-Thompson in 1963) and CompCert [8], an optimizing C compiler developped and proved entirely in Rocq.

Rocq has several components. Its core is called the "kernel" and is responsible for checking the coherence of proofs. The rest of Rocq builds on top of this kernel to provide convenience features, while Rocq's kernel is kept lean to ensure its soundness. One such feature is "tactics langages" (Ltac 1 [9] and its successor Ltac 2, SSReflect [10]). Tactics can be thought of as simple – yet composable – proof gadgets useful to perform a proof step. Rocq can be extended by plugins written in a combination of different languages, OCaml or more recently Elpi [11].

It is common for mathematicians to "identify" isomorphic objects as being equal, in the sense that if a result holds for one object, it also holds for the other. Consider the following inductive definitions of natural numbers:

```
Inductive N_unary : Type := O : N_unary | S (n: N_unary) : N_unary.

Inductive positive : Type :=
  xI : positive -> positive | xO : positive -> positive | xH : positive.
Inductive N_binary : Type := O' : N_binary | Npos : positive -> N_binary.
```

It is not too difficult to find an isomorphism between the two, however results about one definition cannot be immediately obtained from the other, often resulting in time wasted proving morally duplicate statements. In other situations, the two types might be related by something weaker than an isomorphism: there is only a surjection between $\mathbb{Z}$ (*natural numbers*) and $\mathbb{Z}_n$ (*natural numbers modulo* n), yet in some cases we might want to obtain results by specializing lemmas about $\mathbb{Z}$. More generally, many complex cases exist.

Trocq [12] provides a new tactic to rewrite a proof goal into another where some types have been replaced by related types: Trocq translates the goal statement along user-provided relations between types.

During my internship, I worked on improving Trocq in the following aspects:
- improving its codebase to make it easier to use and to make it easier to maintain;
- better understanding its theory and extending its capabilities.

My contributions are:
- proving optimality of the constraints on levels for the arrow and forall types, which were conjectured to be optimal by Trocq's authors [12]. [↪ `Optimality.v`]
- an attempt to introduce two new levels to Trocq's hierarchy. These levels are based on decompositions of the statement that a relation is functional. Quentin Vermande postulated introducing these level would generalize Trocq, following experiments he performed. Unfortunately I could not find definitions for these levels that were stable, in the sense that the hierarchy remains well-behaved. It's unclear whether useful yet well-behaved definitions exist for these new levels.

- attempting to replace the use of equalities in the Trocq hierarchy by other relations. The objective is three-fold: firstly, removing Trocq's dependencies on the functional extensionality and the univalence axiom; secondly, allowing for users defined relations; and finally, considering when weaker relations than equality are required. Our exploratory work theorized the pertinence of introducing a hierarchy of relations based on enriched category theory.
- updating Trocq's supported Rocq version from 8.17 to Rocq 8.20/9.0. This change was made with the objective of making Trocq available in modern Rocq development. [↪ Pull request #46]
- making Trocq support multiple preludes. Using the same codebase, Trocq supports both Rocq's core library and Coq-Hott's library, a remplacement prelude for Rocq development in the setting of HoTT. [↪ Pull request #46]
- unifying how Trocq manage sorts. As for the previous contribution, this is a general codebase improvement to make Trocq easier to maintain. Depending on the prelude loaded, Trocq supports different sorts. My contribution adds a generic API to register supported sorts. [↪ Pull request #52, currently pending for review]

My various contributions resulted in the 0.2.0 release of Trocq [13].

Section II contains information regarding Trocq's theory and the resulting plugin API. Section III details the different ways in which I contributed to Trocq [12] during my internship.

II. Context

Trocq builds upon parametricity, a general technique to relate objects of two similar types. These types are similar in the sense that they have the same "shape", only differing in the free variables they contain. Given these two types, parametricity is a process to automatically derive a relation between them such that the relation "follows" the "shape" of the two types. Trocq builds on two existing parametricity translations: raw parametricity [14–16] and univalent parametricity [17]. Thanks to a careful analysis of type equivalence, Trocq's authors [12] managed to bridge these two kinds of parametricity translations by defining a hierarchy of parametricity translation, which allowed to unify their use. Section II.2 gives an in-depth definition of parametricity.

Thanks to this new hierarchy, Trocq implements proof transfers in a very general setting, improving the state of the art by enabling proof transfers in cases that were previously not possible. A proof transfer can be thought of as obtaining a new proof goal from an initial goal by substituting one or more of the free variables it contains. Section II.1 gives the formal definition of proof transfers. Section II.2 also contains an overview of how prior works utilized parametricity to perform proof transfers.

Section II.3 goes over how type equivalence was disassembled by Trocq's authors to create the hierarchy of parametricities. Finally, Section II.4 details how Trocq ties everything together in a Rocq plugin for proof transfers.

II.1. Proof transfers

Formally, we're trying to accomplish proofs transfers, which can be defined as follows. We give this definition in the Calculus of Constructions, which has the following syntax of dependent terms:

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

where $\square_i$ denote the predicative hierarchy of universes, simply abbreviated $\square$ when the universe level does not matter. Refer to [18] for the (somewhat standard) typing rules, omitted here for brevity. Proof-assistants, such as Rocq, use various extensions of this core calculus.

Given $R_\square : \square \to \square \to \square$, a **proof transfer from a type S to a type T** consists in constructing, from a dependent type $W : T \to \square$, a dependent type $V : S \to \square$ such that there is a proof w as follows, for a *suitable* context Γ and $R :\ S \to T \to \square$:

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

This definition generalizes to transfers over multiple types.

S and T are the related types, $R : S \to T \to \square$ is their relation. W is the result over T we're trying to transfer from V, dependently typed over S. $R_\square$ is how we want statements over S and T to be related: in practice $R_\square A\ B \triangleq B \to A$ is often used, since this way, we can obtain $V\ s$ from $W\ t$ using w.

II.2. Prior state of the art & parametricity

Prior to Trocq, several initiatives trying to accomplish similar goals existed in the literature. For instance, CoqEAL [19] is a Rocq plugin aiming to switch inefficient data structures for efficient ones in algorithms written in Rocq. It is composed of several efficient algorithms as well a framework of "refinements", for changing data representation in a proof. CoqEAL makes use of raw parametricity, which is an instance of a general technique: parametricity. Unfortunately, CoqEAL lacks the ability to deal with general dependent types, and is thus useless in situations involving universal quantifications.

Parametricity is a translation for terms and contexts, denoted $[\![\cdot]\!]$. For contexts, it consists in introducing for each variable $t : T$ of the context a new variable $x' : T'$ and a new relation $x_R : [\![T]\!]\ x\ x'$. For a term t, the notation t' consists in replacing in t all free variables x with the corresponding variable x' introduced during the context translation.

Figure 1 contains the translation of terms, however, the so-called "abstraction theorem" is arguably more important to understand what the parametricity translation *really* consists in.

Theorem *(abstraction theorem)* : If $\Gamma \vdash t : T$, then :
- $[\![\Gamma]\!] \vdash t : T$
- $[\![\Gamma]\!] \vdash t' : T'$
- $[\![\Gamma]\!] \vdash [\![t]\!] : [\![T]\!]\ t\ t'$

According to the abstraction theorem, if we have a type T and $t : T$, then its translation $[\![T]\!]$ is a relation between T and T' and $[\![t]\!]$ is a proof (called the "witness") that t and t' are related by $[\![T]\!]$. For instance, with this in mind, the rule for the product can be read as: "two functions are related when all related inputs have related outputs". The application and lambda-abstraction translation go hand-in-hand.

$$\begin{aligned}
[\![\square_i]\!] &:\ \square_i \to \square_i \to \square_{i+1} \\
[\![x]\!] &= x_R \\
[\![t\ u]\!] &= [\![t]\!]\ u\ u'\ [\![u]\!] \\
[\![\lambda x : A.t]\!] &= \lambda(x : A)(x' : A')(x_R : [\![A]\!]\ x\ x').\ [\![t]\!] \\
[\![\Pi x : A.\ B]\!] &= \lambda(f :\ \Pi_{x:\ A} B)(f' :\ \Pi_{x':\ A'} B'). \\
&\qquad \Pi(x : A)(x' : A')(x_R : [\![A]\!]\ x\ x').\ [\![B]\!]\ (f\ x)\ (f'\ x')
\end{aligned}$$

Figure 1: Parametricity translation for terms

To get a feel for the action of the translation on terms, let's consider its action on the induction principle on $\mathbb{N}$. Recall that the induction principle is defined as:

$$\texttt{N_ind} \triangleq \prod_{P:\ \mathbb{N} \to \square} (P\ 0) \to \left(\prod_{n:\ \mathbb{N}} P\ n \to P\ (S\ n) \right) \to \prod_{n:\ \mathbb{N}} P\ n$$

Its translation is:

$$
\begin{aligned}
&[\![\texttt{N_ind}]\!] \triangleq \lambda\ \texttt{ind}\ \texttt{ind}'.\\
&\qquad \prod(P:\ \mathbb{N}\to\Box)(P':\ \mathbb{N}'\to\Box)\left(P_R:\ \prod_{\substack{n:\,\mathbb{N}\\ n':\,\mathbb{N}'}}(\mathbb{N}_R\ n\ n')\to[\![\Box]\!]\ (P\ n)\ (P'\ n')\right).\\
&\qquad \prod(P_0:\ P\ 0)(P_0':\ P'\ 0').\ (P_R\ 0\ 0'\ 0_R\ P_0\ P_0')\to\\
&\qquad \prod\left(P_{\text{rec}}:\prod_{n:\,\mathbb{N}}(P\ n)\to P\ (S\ n)\right)\left(P_{\text{rec}}':\ \prod_{n':\,\mathbb{N}'}(P'\ n')\to P'\ (S'\ n')\right).\\
&\qquad\quad \left(\prod_{\substack{n:\,\mathbb{N}\\ n':\,\mathbb{N}'\\ n_R:\,\mathbb{N}_R\ n\ n'}}\ \prod_{\substack{H_n:\ P\ n\\ H_{n'}:\ P'\ n'}}(P_R\ n\ n'\ n_R\ H_n\ H_n')\to\right.\\
&\qquad\qquad \left. P_R\ (S\ n)\ (S'\ n')\ (S_R\ n\ n'\ n_R)\ (P_{\text{rec}}\ n\ H_n)\ (P_{\text{rec}}'\ n'\ H_n')\right)\to\\
&\qquad \prod(n:\ \mathbb{N})(n':\ \mathbb{N}')(n_R:\ N_R\ n\ n').\\
&\qquad\quad P_R\ n\ n'\ n_R\ (\texttt{ind}\ \ P\ P_0\ P_{\text{rec}}\ n)\ (\texttt{ind}'\ P'\ P_0'\ P_{\text{rec}}'\ n')
\end{aligned}
$$

$[\![\texttt{N_ind}]\!]$ is a lambda-term taking two induction proofs `ind` : `N_ind` and `ind'` : `N_ind'`. The return value closely mimics `N_ind`, but where products duplicate their quantification with a additional primed version of the object and with a proof of relation between the object and its primed counterpart.

It can be observed that Figure 1 does not contain a definition for the translation of $\Box_i$. Actually, the definition of $[\![\Box_i]\!]$ defines the type of relations we obtain after applying parametricity. Indeed, for all typed terms t there is a level i such that $t : \Box_i$, thus by the abstraction theorem, $[\![t]\!] : [\![\Box_i]\!]\ t\ t'$. Below are two common settings of parametricity, each defining $[\![\Box_i]\!]$.

Raw parametricity $[\![\Box_i]\!]$ is defined as:

$$[\![\Box_i]\!] \triangleq \lambda(A\ B : \Box_i), A \to B \to \Box_i$$

The relation obtained between T and T' via parametricity can be instantiated with any relation.

For proof transfers, raw parametricity works well for statements about a closed datatype. For instance, we might want to prove that multiplying 15 and 46 as `N_unary` numbers is the "same as" multiplying 15 and 46 as `N_binary` numbers. The notion of "being the same" for objects of types `N_unary` and `N_binary` are given by $[\![\texttt{N_unary}]\!] \triangleq \texttt{N_unary}_R$. If $\downarrow$: `N_unary` $\to$ `N_binary` is the isomorphism between these two types, $\texttt{N_unary}_R$ can be taken as:

$$\texttt{N_unary}_R\ n\ n' \triangleq (\downarrow n = n')$$

Applying the abstraction theorem to $\text{mult}_{\texttt{N_unary}}\ 15_{\texttt{N_unary}}\ 46_{\texttt{N_unary}}$ gives that $[\![\text{mult}_{\texttt{N_unary}}\ 15_{\texttt{N_unary}}\ 46_{\texttt{N_unary}}]\!]$ is of type:

$$[\![\texttt{N_unary}]\!]\ (\text{mult}_{\texttt{N_unary}}\ 15_{\texttt{N_unary}}\ 46_{\texttt{N_unary}})\ (\text{mult}_{\texttt{N_binary}}\ 15_{\texttt{N_binary}}\ 46_{\texttt{N_binary}})$$

Or, by definition of $\texttt{N_unary}_R$:

$$\downarrow(\text{mult}_{\texttt{N_unary}}\ 15_{\texttt{N_unary}}\ 46_{\texttt{N_unary}}) = (\text{mult}_{\texttt{N_binary}}\ 15_{\texttt{N_binary}}\ 46_{\texttt{N_binary}})$$

Unfortunately, raw parametricity fails to perform proof transfer on stronger statements that involve quantification: obtaining a relation between a type T and T' via raw parametricity will not provide us with more information other than that it's a relation.

Univalent parametricity In this setting, we maintain additional data along

with the relation. Another translation is defined: $[\cdot]$. On contexts, it operates similarly to $[\![\cdot]\!]$.

On terms, the translation of T consists in a dependent triple of type $\boxdot\ T\ T'$, with $\boxdot$ defined as[1]:

$$\boxdot \triangleq \lambda T\ T'. \sum (R : T \to T' \to \square)(e : T \simeq T') \prod_{\substack{t:\ T \\ t':\ T'}} (R\ t\ t') \simeq \left(t = e_{\downarrow}\ t'\right)$$

where $A \simeq B$ denotes an equivalence between the types A and B. An equivalence is given as a function $e_{\uparrow} : B \to A$, a function $e_{\downarrow} : A \to B$ and a proof of a coherence property between $e_{\uparrow}$ and $e_{\downarrow}$, ensuring proof-irrelevance. Equivalences and their significance will be discussed in greater details below.

For a term T, the definition of $[T]$ is such that its first projection will be $[\![T]\!]$. The definition of $[\![T]\!]$ for terms is the same as the general parametricity setting we presented, with the exception that for variables x, $[\![x]\!]$ is the first component of x_R. $[\![\square_i]\!]$ is defined as $\boxdot_i$.

For the other components of $[T]$, we define them by induction on T in Figure 2. As for $[\![\cdot]\!]$, the translation of applications and lambda-abstractions work hand-in-hand and should be understood in that regard.

$$\begin{aligned}
[\square_i] &= \left(\boxdot_i ; \mathrm{id}_{\square_i} ; \mathrm{univ}_{\square_i}\right) \\
[x] &= \ x_R \\
[t\ u] &= \ [t]\ u\ u'\ [u] \\
[\lambda x : A.t] &= \ \lambda(x : A)(x' : A')(x_R : [\![A]\!]\ x\ x').\ [t] \\
[\Pi x : A.\ B] &= (\ \lambda f f'.\ \Pi(x : A)(x' : A')(x_R : [\![A]\!]\ x\ x').\ [\![B]\!]\ (f\ x)\ (f'\ x'); \\
&\quad\ \mathrm{Equiv}_{\Pi}\ [A]\ [B];\ \mathrm{univ}_{\Pi}\ [A]\ [B])
\end{aligned}$$

where the relation and equivalence parameters of univ_{Π} are implicit and $(\ \cdot\ ;\ \cdot\ ;\ \cdot\ )$ denotes a dependent triple.

Figure 2: Univalent parametricity translation for terms

Equiv_{Π} constructs an equivalence $\left(\prod_{x:\ A} B\right) \simeq \left(\prod_{x':\ A'} B'\right)$ from the equivalences $A \simeq A'$ and $B \simeq B'$. univ_{Π} relate this equivalence with the relation $\left[\!\left[\prod_{x:\ A} B\right]\!\right]$. $\mathrm{id}_{\square_i} : \square_i \simeq \square_i$ is the identity equivalence and $\mathrm{univ}_{\square_i}$ relates it to $\boxdot_i$. When expanded out, $\mathrm{univ}_{\square_i}$ is exactly univalence for Trocq's alternative definition of equivalence.

Compared to raw parametricity, a translated term $[T]$ is much richer: not only it contains a relation $[\![T]\!] : T \to T' \to \square$ between T and T', it also features an equivalence e between T and T' as well as a proof of coherence between $[\![T]\!]$ and the equivalence e. The equivalence e provides transfer functions $e_{\uparrow} : T' \to T$ and $e_{\downarrow} : T \to T'$ which can be used to perform the proof transfer.

If univalent parametricity is useful to perform proof transfer, it is only restricted in cases where we have a full equivalence between objects. Intuitively, an equivalent exists when objects are isomorphic.

[1]Having access to a dependent product requires a stronger calculus compared to the setting initially presented in this section: for instance, the calculus of constructions with inductive types or Homotopy Type Theory.

Thus, we would not be able to use univalence parametricity in weaker cases: for instance, to transfer a result from $\mathbb{Z}$ to $\mathbb{Z}_n$, where there is no equivalence between these two objects.

II.3. HoTT, Univalence and disassembling type equivalence

Univalence parametricity works well when there is an equivalence between types. To allow proof transfers in weaker cases, Trocq uses various "levels" of relations between two types T and T': a spectrum which goes from only having a relation $R : T \to T' \to \square$ as in raw parametricity, to having as much data alongside the relation R as in univalent parametricity.

II.3.1. Homotopy Type Theory and the Univalence axiom

Trocq's different "levels" originate from a clever disassembly of equivalence. To perform this disassembly, Trocq uses the setting of Hotomopy Type Theory (HoTT) [20]. HoTT is a recent type theory aiming to be a development of mathematics based on type-theory, using ideas from higher-category theory and homotopy theory. Using HoTT is natural since univalence is a core part of it.

For our use case in this section, we can omit the deep reasons as to why HoTT is a satisfying framework to set ourselves in and naively view HoTT as the traditional calculus of inductive constructions, a calculus of constructions with inductive types, with an additional axiom: the univalence axiom.

In HoTT, an equivalence between two types $A, B : \square$ is defined[2] as:

$$A \simeq B \triangleq \sum (\uparrow : B \to A)(\downarrow : A \to B)(\eta : \downarrow \circ \uparrow \sim \mathrm{id}_B)(\varepsilon : \uparrow \circ \downarrow \sim \mathrm{id}_A) \prod_{x:A} \downarrow (\eta\ x) = \varepsilon(\downarrow x)$$

where $f \sim g \triangleq \prod_x f\ x = g\ x$ is point-wise equality of functions, and if $f : \prod_{x:\ A} B$ and $e : a \underset{A}{=} a'$, $f\ e$ is a proof of equality over B of $f\ a$ and $f\ a'$.

The univalence axiom is: for all $A, B : \square$, $(A \simeq B) \simeq (A = B)$. Moreover, the $(A = B) \to (A \simeq B)$ function from the equivalence of the axiom is the function we could already define by eliminating on the equality. The univalence axiom states that equality on types is equivalent to type equivalence. This axiom is useful to obtain equalities between some types we would not otherwise be able to proof equal by instead proving an equivalence. Remark: the univalence axiom implies functional extensionality:

$$\text{for all } A : \square,\ B : A \to \square, \text{ for all } f, g : \Pi_{a:A}(Ba),\ (f \sim g) \simeq (f = g)$$

II.3.2. Disassembling type equivalence

Cohen, Crance and Mahboubi [12] [↪ lemma `uparam_equiv`, `Uparam.v`] proved that type of equivalences between A and B, $A \simeq B$, is equivalent to the following type:

$$\sum_{R:\ A \to B \to \square} \mathrm{IsUmap}(R) \times \mathrm{IsUmap}(R^{-1})$$

where $R^{-1} \triangleq \lambda(b : B)(a : A).\ R\ a\ b$ flips the order of the arguments of R and $\mathrm{IsUmap}(R)$, with $R : A \to B \to \square$, is defined as:

$$\mathrm{IsUmap}(R) \triangleq \sum \begin{cases} \texttt{map} : A \to B \\ \texttt{map_in_R} : \Pi_{(a:\ A)(b:\ B)}(\texttt{map}\ a = b) \to R\ a\ b \\ \texttt{R_in_map} : \Pi_{(a:\ A)(b:\ B)}(R\ a\ b) \to \texttt{map}\ a = b \\ \texttt{R_in_mapK} : \Pi_{\substack{a:\ A \\ b:\ B}} (\texttt{map_in_R}\ \ a\ b) \circ (\texttt{R_in_map}\ \ b\ a) \sim \mathrm{id} \end{cases}$$

[2] Actually, equivalences can be defined in many different but equivalent ways in HoTT. The half adjoint equivalence definition is given here. See chapter 4 of the HoTT book [20] for more information.

With the objective of creating a hierarchy, this disassembly of type equivalence hints at the definition of a lattice of levels: it can be observed that in the case of raw parametricity, we have none of `map`, `map_in_R`, `R_in_map` and `R_in_mapK` for both R and R^{-1}. For univalent parametricity, we would have all of them.

For IsUmap, Trocq's resulting level lattice is $\{0, 1, 2a, 2b, 3, 4\}$, with the following relations:

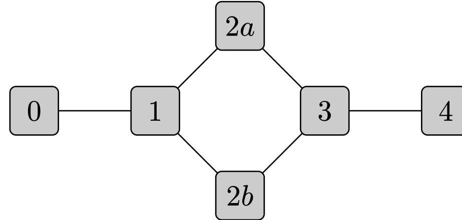


Figure 3: Trocq's lattice of levels

For each level M of this lattice, $\texttt{Map}M(R)$ is a record (a dependent tuple):

- $\texttt{Map}0(R)$ is the empty record;
- $\texttt{Map}1(R)$ contains `map`;
- $\texttt{Map}2a(R)$ contains `map`, `map_in_R`;
- $\texttt{Map}2b(R)$ contains `map`, `R_in_map`;
- $\texttt{Map}3(R)$ contains `map`, `map_in_R`, `R_in_map`;
- $\texttt{Map}4(R)$ contains all 4 fields;

IsUmap is `Map4`. Having defined $\texttt{Map}M$ over the lattice $\{0, 1, 2a, 2b, 3, 4\}$, $\texttt{Param}MN$ can be defined over the lattice $\mathcal{A} \triangleq \{0, 1, 2a, 2b, 3, 4\}^2$ as follows:

$$\forall (M, N) \in \mathcal{A}, \;\; \texttt{Param}MN\ A\ B = \sum_{R:\ A \to B \to \square} \texttt{Map}M(R) \times \texttt{Map}N(R^{-1})$$

To refer to the record elements of $\texttt{Map}N(R^{-1})$ in the definition above, we'll refer to its `map` projection as `comap` and rename the other fields accordingly. Moreover, we'll denote the relation R of $P : \texttt{Param}MN\ A\ B$ as $\mathrm{rel}(P)$.

This hierarchy proves useful because it can represent exactly known concepts of relations between types:

- the level $(0, 0)$ corresponds to only having a relation between the two types. Having no further results on this relation, a parametricity witness at level $(0, 0)$ does not relate its two types at all.
- the level $(4, 4)$ is equivalent to type equivalence;
- the level $(4, 2a)$ between A and B is equivalent to having a retraction $A \to B$: that is to say, a surjective map $A \to B$ with an explicit partial left inverse $B \to A$. The relation is in this case the *graph* of the map $A \to B$.
- the level $(4, 2b)$ between A and B is equivalent to having a section $A \to B$: that is to say, a injective map $A \to B$ with an explicit partial right inverse $B \to A$. The relation is in this case the *graph* of the map $A \to B$.

II.4. Trocq

II.4.1. Proof transfer

It can be observed that performing a proof transfer of a property $P : T \to \square$ from T to T' can be achieved by constructing a term of type $\texttt{Param}01\ T\ T'$. This term contains $\texttt{comap} : T' \to T$ which we can use to perform the transfer.

II.4.2. Trocq's parametricity translation, presented in sequent-style

Constructing such a term will require some structure, expressed as terms of types $\texttt{Param}\alpha$, on the free variables it contains.

With the inspiration from parametricity, the construction of $\texttt{Param}\alpha\ T\ T'$ can be performed via induction on T. However, contrary to "classical" parametricity, due to the multiplicity of levels, a term can be translated in multiple ways. Indeed, what relation should be used for the term $p_\square^\alpha : \texttt{Param}\alpha\ \square_i\ \square_i$? Univalent parametricity hints to use $\texttt{Param}\beta\ \square_j\ \square_j$ for a level $\beta \in \mathcal{A}$. As a result, Trocq actually consider all terms (whenever they exist) $p_\square^{\alpha,\beta} : \texttt{Param}\beta\ \square_i\ \square_i$ such that $\mathrm{rel}\left(p_\square^{\alpha,\beta}\right) \triangleq \texttt{Param}\alpha$.

Thus, to guide parametric translation, we consider the translation of $t : T$ at "level A", where A is the type T *annotated* with level information. Moreover, t is also annotated with level information. The annotated terms are like their unannotated counterparts, but where universes are annotated with a level $\alpha \in \mathcal{A}$ which provide information about which $\texttt{Param}$ relation should be used after translation.

To express the translation rules, we define the relation $t\ @\ A \sim t' \because p$, which reads as "$t$ at level A is related to t' via p". The translation rules are given in Figure 4. $D_\square \subseteq \mathcal{A}$ is the set of pairs (α, β) such that $p_\square^{\alpha,\beta}$ can be defined. Similarly, for $\gamma \in \mathcal{A}^2$, $D_\Pi(\gamma) \subseteq \mathcal{A}$ is the set of pairs (α, β) such that p_Π^γ exists (where, for now, α and β are implicit). $p_\Pi^\gamma\ A_R\ B_R$ is a term of type $\texttt{Param}\gamma\ (\Pi_{x:A} B)\ (\Pi_{x:A'} B')$ where its relation is:

$$\mathrm{rel}(p_\Pi^\gamma\ A_R\ B_R) \triangleq \lambda f f'. \prod (x :\ A)(x' :\ A')(x_R : \mathrm{rel}(A_R)\ x\ x').\ \mathrm{rel}(B_R)\ (f\ x)\ (f'\ x')$$

The TrocqConv rule exists to represent the subtyping rules of the annotated terms. Since $\mathcal{A}$ is a lattice, we have an obvious subtyping relation between annotated terms, $\leq$. If $A \leq B$, $\underset{B}{\overset{A}{\Downarrow}}$ consists in the weakening of parametricity witnesses. Details about the definitions of these objects can be found in section 6.2 of [12].

$$\frac{(\alpha,\beta) \in D_\square}{\Delta \vdash \square_i^\alpha\ @\ \square_{i+1}^\beta \sim \square_i^\alpha \because p_{\square_i}^{\alpha,\beta}}\ (\textsc{TrocqSort})$$

$$\frac{\Delta \vdash t\ @\ \Pi_{x:A} B \sim t' \because t_R \qquad \Delta \vdash u\ @\ A \sim u' \because u_R}{\Delta \vdash t\ u\ @\ B[x := u] \sim t'\ u' \because t_R\ u\ u'\ u_R}\ (\textsc{TrocqApp})$$

$$\frac{\Delta \vdash A\ @\ \square_i^\alpha \sim A' \because A_R \qquad \Delta, x\ @\ A \sim x' \because x_R \vdash t\ @\ B \sim t' \because t_R}{\Delta \vdash \lambda(x : A).\ t\ @\ \Pi_{x:A} B \sim \lambda(x' : A').\ t' \because \lambda x\ x'\ x_R.\ t_R}\ (\textsc{TrocqLam})$$

$$\frac{\begin{array}{c}(\alpha,\beta) \in D_\Pi(\gamma) \qquad \Delta \vdash A\ @\ \square_i^\alpha \sim A' \because A_R \\ \Delta, x\ @\ A \sim x' \because x_R \vdash B\ @\ \square_i^\beta \sim B' \because B_R\end{array}}{\Delta \vdash \Pi_{x:A} B\ @\ \square^\gamma \sim \Pi_{x':A'} B' \because p_\Pi^\gamma\ A_R\ B_R}\ (\textsc{TrocqPi})$$

$$\frac{\Delta \vdash t\ @\ A \sim t' \because t_R \qquad \Delta \vdash A \leq B}{\Delta \vdash t\ @\ B \sim t' \because \underset{B}{\overset{A}{\Downarrow}} t_R}\ (\textsc{TrocqConv})$$

Figure 4: Trocq's translation for terms

II.4.3. Constraints and dependencies of levels

It turns out that $D_\square = \{0, 1, 2a\}^2 \times \mathcal{A}$. However, if we allow ourselves to use the univalence axiom, then $D_\square = (\{0, 1, 2a\}^2 \times \mathcal{A}) \cup (\mathcal{A} \times \{(4, 4)\})$.

What is $D_\Pi(\gamma)$ for $\gamma \in \mathcal{A}$? A first easy observation is that, if $(\alpha, \beta) \in D_\Pi(\gamma)$, then for all $(\alpha, \beta) \leq (\alpha', \beta')$, we also have $(\alpha', \beta') \in D_\Pi(\gamma)$ because something at a level α' can be weakened to be at level α. Moreover, $p_\Pi^{(M,N)}\ A_R\ B_R$ is a dependent pair composed of a relation, a term of type $\texttt{Map}M(R)$ and a term of type $\texttt{Map}N(R^{-1})$. All $p_\Pi^\gamma\ A_R\ B_R$ terms have the same relation, so we

can recompose $p_\Pi^{(M,N)}$ from $p_\Pi^{(0,N)}$ and $p_\Pi^{(M,0)}$. Furthermore, it's easy to obtain $p_\Pi^{(N,M)}$ out of $p_\Pi^{(M,N)}$. From all of these observations, it is thus seems sufficient to specify D_Π as a function from $\mathcal{A}$ to $\mathcal{A}^2$ on the left and right components of its domain. Table 1 list the constraints achieved by [12], although their optimality was only theorized. Functional extensionality is also in some cases required.

m	$D_\Pi(m,0)_1$	$D_\Pi(m,0)_2$
0	$(0,0)$	$(0,0)$
1	$(0,2a)$	$(1,0)$
$2a$	$(0,4)$	$(2a,0)$
$2b$ *(F)*	$(0,2a)$	$(2b,0)$
3 *(F)*	$(0,4)$	$(3,0)$
4 *(F)*	$(0,4)$	$(4,0)$

m	$D_\to(m,0)_1$	$D_\to(m,0)_2$
0	$(0,0)$	$(0,0)$
1	$(0,1)$	$(1,0)$
$2a$	$(0,2b)$	$(2a,0)$
$2b$ *(F)*	$(0,2a)$	$(2b,0)$
3 *(F)*	$(0,3)$	$(3,0)$
4 *(F)*	$(0,4)$	$(4,0)$

Axiom requirements: *(F)* for functional extensionality

Table 1: Constraints for the forall and arrow types

In type theory, the arrow type $A \to B$ is simply the product type $\Pi_{x:A}B$ where B does not dependent on x. However, it turns out the constraints are in some cases weaker for the arrow type. Table 1 lists the constraints for the arrow type; cells in red indicate the changes compared to D_Π. This realization leads to the introduction of a new rule specific to the arrow type; see Figure 5.

$$\frac{(\alpha,\beta) = D_\to(\gamma) \qquad \Delta \vdash A \mathbin{@} \square_i^\alpha \sim A' \because A_R \qquad \Delta \vdash B \mathbin{@} \square_i^\beta \sim B' \because B_R}{\Delta \vdash \Pi_{x:A}B \mathbin{@} \square_i^\delta \sim \Pi_{x':A'}B' \because p_\Pi^\gamma\ A_R\ B_R}\ (\textsc{TrocqArrow})$$

Figure 5: The TrocqArrow rule

II.4.4. Trocq as a Rocq library

Trocq is a Rocq plugin. Having its theory built in HoTT, Trocq was developed based on Coq-HoTT [21], a Rocq library that implements the Homotopy Type Theory. For technical reasons, Coq-HoTT is incompatible with Rocq's standard library, Stdlib. Thus, Rocq must be configured with the `no-init` flag, which disables loading Rocq's prelude. HoTT does not have an impredicative sort.

However, the concrete usecases where Trocq could be useful probably are not in the setting of HoTT. To be useful, Trocq provides a variant compatible with Rocq's Stdlib, in a separate `prop` branch. Contrary to HoTT enthusiasts, the use of the functional extensionality axiom is less accepted along Rocq users, and even more so for the univalence axiom.

Trocq adds commands to register the available axioms, see Listing 1. Trocq will complain whenever it needs an axiom that was not registered.

```
Trocq Register Univalence u.
Trocq Register Funext f.
```

Listing 1: Trocq's vernacular commands for registering available axioms

When using Trocq's Stdlib variant, Trocq must also be able to handle `Prop`. To do so, the associated Trocq variant considers an additional rule, TrocqProp (Listing 2). Similarly to $\square$, `Prop` is also annotated with a Trocq level $\alpha \in \mathcal{A}$. $D_{\texttt{Prop}} = \{0,1,2a\}^2 \times \mathcal{A}$.

$$\frac{(\alpha,\beta) \in D_{\texttt{Prop}}}{\Delta \vdash \texttt{Prop}^\alpha \mathbin{@} \texttt{Prop}^\beta \sim \texttt{Prop}^\alpha \because p_{\texttt{Prop}}^{\alpha,\beta}}\ (\text{TrocqProp})$$

with $p_{\texttt{Prop}}^{\alpha,\beta} : \texttt{Param}\alpha\ \texttt{Prop}\ \texttt{Prop}$ and $\mathrm{rel}\left(p_{\texttt{Prop}}^{\alpha,\beta}\right) = \texttt{Param}\beta\ \texttt{Prop}\ \texttt{Prop}$

Listing 2: Trocq's parametric translation for the `Prop` sort

Parametric relations between objects can be registered to Trocq via vernacular commands. For instance, if `NR` : `Param44 N_unary N_binary`, the vernacular command `Trocq Use NR.` can be issued to inform Trocq of its existence.

Finally, Trocq implements the `trocq` tactic to actually perform a proof transfer.

II.4.5. Implementation

Trocq is written in Elpi [11] using its associated Rocq-Elpi plugin. Elpi stands for "Embeddable λ-Prolog Interpreter". It is a Turing-complete programming language inspired from λ-Prolog. Elpi simplifies the development of Rocq plugins by allowing its users to write Elpi code directly in Rocq (hence the "Embeddable" in Elpi's name), without an extra compilation step that would, for instance, be required when writing an Ocaml Rocq plugin.

Trocq uses Elpi in various ways:

- for state management thanks to Elpi databases [↪ `Database.v`] [↪ `elpi/database.elpi`]
- for automatic Rocq code generation [↪ `elpi/generation/`]
- to implement its `trocq` tactic [↪ `elpi/tactic.elpi`].

III. Contributions

This section presents how I contributed to Trocq during my internship.

III.1. Proving the optimality of constraints for forall & arrow types

Let $A : \square$ and $B : \square$ *(resp.* $B : A \to \square$*)*.

Table 1 gives the constraints achieved by Trocq on the levels of the input types A and B needed to construct output type $A \to B$ *(resp.* $\Pi_{x:A} B\ x$*)*. Given the level $\gamma \in \mathcal{A}$ wanted for $p^{\gamma}_{\to} : \texttt{Param}\gamma\ A_R\ B_R$ *(resp. for* $p^{\gamma}_{\Pi} : \texttt{Param}\gamma\ A_R\ B_R$*)*, $D_{\to}(\gamma)$ *(resp.* $D_{\Pi}(\gamma)$*)* is a pair (α, β) giving the type needed on A_R and B_R:

$$A_R : \texttt{Param}\alpha\ A\ A'$$
$$B_R : \texttt{Param}\beta\ B\ B'$$
$$(\textit{resp. } B_R : \prod_{\substack{x:\ A \\ x':\ A'}} (x_R : A_R\ x\ x').\ \ \texttt{Param}\beta\ (B\ x)\ (B'\ x'))$$

Trocq's authors obtained this table by attempting to construct the terms $p^{\gamma}_{\to}$ and p^{γ}_{Π}, seeing the elements of A_R and B_R they ended up using assuming these two parametric relations had level $(4, 4)$.

Of course, Table 1 provides as upper bound on the optimality levels. Moreover, because in type-theory, the arrow type is an instance of the product type, we have :

$$\forall \gamma \in \mathcal{A},\ D_{\to}(\gamma) \leq D_{\Pi}(\gamma)$$

Stating the obvious, if $D_{\to}(\gamma)$ is proven optimal and $D_{\Pi}(\gamma) = D_{\to}(\gamma)$ in Table 1, then $D_{\Pi}(\gamma)$ is also optimal. The strategy is thus to prove optimality of $D_{\to}$ in Table 1, and then prove optimality of $D_{\Pi}(m, 0)_1$ for $m \in \{1, 2b, 3\}$.

Since constraints $D_{\to}$ *(resp.* D_{Π}*)* can be given separately as the image of each component on the domain, proving the optimality of Table 1 would thus mean finding terms A, A', B, B', A_R at level α and B_R at level β such that the type p^{γ}_{Π} or the type $p^{\gamma}_{\to}$ is inhabited. In this case, $(\alpha, \beta) \leq D_{\to}(\gamma)$ *(resp.* $(\alpha, \beta) \leq D_{\Pi}(\gamma)$*)*.

The optimality of Table 1 was proved in `Optimality.v` [↪ source] which is now part of Trocq. This Rocq proof file contains 30 theorems for each case, along with accompagning definitions of the objects. I will present three examples of optimality proofs.

Firstly, let us denote $R_\to$ and R_Π the relations of, respectively, $p^\gamma_\to$ and p^γ_Π:

$$R_\to\ A\ A'\ A_R\ B\ B'\ B_R \triangleq \lambda f f'.\ \prod_{\substack{a:\,A\\ a':\,A'}} (A_R\ a\ a') \to B_R\ (f\ a)\ (f'\ a')$$

$$R_\Pi\ A\ A'\ A_R\ B\ B'\ B_R \triangleq \lambda f f'.\ \prod_{\substack{a:\,A\\ a':\,A'\\ a_R : A_R\ a\ a'}} B_R\ a\ a'\ a_R\ (f\ a)\ (f'\ a')$$

$D_\to(1,0)_1 \geq (0,1)$ We'll prove $D_\to(1,0)_1 > (0,0)$ by proving that, for some types A, A' such that there is a parametricity relation $A_R : \texttt{Param00}\ A\ A'$ and for some types B, B' such that there is a parametricity relation $B_R : \texttt{Param10}\ B\ B'$ such that the type

$$\prod_{R:\,\texttt{Param10}\ (A \to B)\ (A' \to B)} \mathrm{rel}(R) = R_\to\ A\ A'\ A_R\ B\ B'\ B_R$$

has no inhabitants. We will suppose it is inhabited and prove `False`, the empty type.

Remark: here, we state this property using HoTT's equality type ($=$) rather than definitional equality ($\triangleq$) simply because stating it is easier in the corresponding Rocq proof. Since definitional equality is "stronger" than the equality type and that the proof works nonetheless, this is not an issue.

`Map0` is simply the empty record, so we can use any types A, A' and relation $A_r :\ A \to A' \to \square$ between then, since $A_R : \texttt{Param00}\ A\ A'$ such that $\mathrm{rel}(A_R) \triangleq A_r$ will always exist.

$\texttt{Param10}\ B\ B'$ only consists in an relation $B_r : B \to B' \to \square$ as well as a $\texttt{map} : B \to B'$.

For B and B', we'll use `False`. $\mathrm{id}_{\texttt{False}} : \texttt{False} \to \texttt{False}$ can be used for the `map`, and $\lambda\ b\ b'.\ \texttt{Unit}$ as $\mathrm{rel}(B_R)$. For A and A', we'll use `False` and `Unit` with $\lambda\ a\ a'.\ \texttt{Unit}$ as $\mathrm{rel}(A_R)$.

So, we're given a witness of:

$$\texttt{Param10}\ (A \to B)\ (A' \to B') = \texttt{Param10}\ (\texttt{False} \to \texttt{False})\ (\texttt{Unit} \to \texttt{False})$$

This witness includes a field $\texttt{map} :\ (\texttt{False} \to \texttt{False}) \to (\texttt{Unit} \to \texttt{False})$, which can simply be instanciated with $\mathrm{id}_{\texttt{False}}$ and the constructor of `Unit` to obtain a term of type `False`.

This proof was quite easy and we did not even use $R_\to$. Its Rocq formalization can be found in `Optimality.v` [↪ theorem `D1_arrow_left_is_gt0`]. All proofs will, in essence, be similar: using the right objects to obtain a witness of the type and then deducing a contradiction.

$D_\to(3,0)_1 \geq (0,3)$ We'll prove that $D_\to(3,0)_1 > (0,2a)$ and $D_\to(3,0)_1 > (0,2b)$. For each proof, we must find A, A' such that there is $A_R : \texttt{Param02}a\ A\ A'$ (resp. $\texttt{Param02}b\ A\ A'$) but such that $\texttt{Param03}\ A\ A'$ is an empty type.

1. $D_\to(3,0)_1 > (0,2b)$:

   We'll use the fact that we have a $\texttt{Param2}b\texttt{2}b$ parametric relation when using the empty relation $R_{\texttt{False}}$, with any two types A, A' of which we have terms. Indeed, $\texttt{Map2}b(R_{\texttt{False}})$ is the record with two fields: $\texttt{map} : A \to A'$ and $\texttt{R_in_map} : \Pi(a : A)(a' : A').(R_{\texttt{False}}\ a\ a') \to \texttt{map}\ a = a'$. Because we know a term of A', `map` can be constructed. Moreover, assuming a witness of `False`, we can prove anything, so `R_in_map` can also be constructed.

Of course, we will not have a map_in_R : $\Pi_{\substack{a:A \\ a':A'}}$ (map $a = a'$) $\to$ ($R_{\texttt{False}}$ a a') because we use the empty relation.

For the sake of simplicity, let's use $A = A' =$ Unit.

Regarding B and B', we'll use $B = B' =$ Bool and the identity parametricity relation between them.

We obtain a term of type Param33 (Unit $\to$ Bool) (Unit $\to$ Bool). It contains a field R_in_map : $\Pi_{f,g:\ \texttt{Unit} \to \texttt{Bool}}(R_\to\ f\ g) \to$ (map $f = g$).

Having carefully chosen $R_{\texttt{False}}$ as the relation, we can see that $R_\to\ f\ g$ holds for any two functions f and g. Thus, we obtain that for any two f and g, map $f = g$. This immediately leads to a contradiction when taking $f_\bot$ and $f_\top$ as the two constants functions:

$$\top = f_\top\ \texttt{tt} = (\texttt{map}\ f_\top)\ \texttt{tt} = f_\bot\ \texttt{tt} = \bot$$

My formalization of this proof can be found in Optimality.v [↪ theorem D3_arrow_left_isnt_2b].

2. $D_\to(3,0)_1 > (0,2a)$:

   We use $A = A' =$ Bool along with the parametric relation A_R : Param2a2a A A' with the "complete" relation $\mathrm{rel}(A_R)\ a\ a' \triangleq$ Unit, $B = B' =$ Bool with the identity parametricity relation.

   We then obtain immediately a contradiction.

   My formalization of this proof can be found in Optimality.v [↪ theorem D3_arrow_left_isnt_2a].

$D_\Pi(4,0)_1 \geq (4,0)$ This proof is perhaps the most complicated. As always, we'll prove it by proving that $D_\Pi(4,0)_1 > (3,0)$.

We are looking for A, A' and A_R : Param03 A A' such that we do not have a Param44 A A' relation. An easy way to obtain such a type is to take $A = A' =$ Unit and the relation $\mathrm{rel}(A_R) \triangleq \lambda aa'.$ Bool. A_R can be built from the identity maps for map and comap. We always have a witness of $\mathrm{rel}(A_R)\ a\ a'$, so map_in_R and R_in_comap are easy to obtain using a constant boolean. Moreover, since Unit only has one witness, R_in_map and comap_in_R also exist. But we will not have an Param44 relation because Bool has two possible witnesses, while for all a, a' : Unit, map_in_R a b will be a constant. So, we cannot hope to obtain R_in_mapK of type:

$$\prod_{\substack{a:A \\ b:B}} (\texttt{map_in_R}\ a\ b) \circ (\texttt{R_in_map}\ b\ a) \sim \mathrm{id}$$

Regarding B, B' and their relation, we'll use $B = B' = \square$ and $p_\square^{(4,0),(4,4)}$.

We thus obtain R : Param40 (Unit $\to \square$) (Unit $\to \square$). R contains the following fields:

$$\texttt{map} : (\texttt{Unit} \to \square) \to \texttt{Unit} \to \square$$

$$\texttt{R_in_mapK} : \prod_{f,g:\ \texttt{Unit} \to \square} (\texttt{map_in_R}\ f\ g) \circ (\texttt{R_in_map}\ f\ g) \sim \mathrm{id}$$

$$\begin{aligned}\texttt{map_in_R}\ f\ g : \texttt{map}\ f = g \to \mathrm{rel}(R)\ f\ g \\ \triangleq \texttt{map}\ f = g \to \prod_{a,a' :\, \texttt{Unit}} \texttt{Bool} \to B_R\ (f\ a)\ (g\ a') \\ \simeq \texttt{map}\ f = g \to \texttt{Unit} \to \texttt{Unit} \to \texttt{Bool} \to \texttt{Param}44\ (f\ \texttt{tt})\ (g\ \texttt{tt})\end{aligned}$$

$$\begin{aligned}\texttt{R_in_map}\ f\ g : \mathrm{rel}(R)\ f\ g \to \texttt{map}\ f = g \\ \simeq (\texttt{Unit} \to \texttt{Unit} \to \texttt{Bool} \to \texttt{Param}44\ (f\ \texttt{tt})\ (g\ \texttt{tt})) \to \texttt{map}\ f = g\end{aligned}$$

We know that $\texttt{Param}44$ is equivalent to type equivalence. Using Univalence to obtain that $\texttt{Param}44$ is equal to $\simeq$, we obtain:

$$\texttt{map_in_R}\ f\ g : \texttt{map}\ f = g \to \texttt{Unit} \to \texttt{Unit} \to \texttt{Bool} \to (f\ \texttt{tt}) \simeq (g\ \texttt{tt})$$

$$\texttt{R_in_map}\ f\ g : (\texttt{Unit} \to \texttt{Unit} \to \texttt{Bool} \to (f\ \texttt{tt}) \simeq (g\ \texttt{tt})) \to \texttt{map}\ f = g$$

Instanciating these terms with the constants functions $f, g \triangleq \lambda x.\ \texttt{Bool}$ gives:

$$\texttt{map_in_R}\ f\ g : \texttt{map}\ f = g \to \texttt{Unit} \to \texttt{Unit} \to \texttt{Bool} \to \texttt{Bool} \simeq \texttt{Bool}$$

$$\texttt{R_in_map}\ f\ g : (\texttt{Unit} \to \texttt{Unit} \to \texttt{Bool} \to \texttt{Bool} \simeq \texttt{Bool}) \to \texttt{map}\ f = g$$

In HoTT [20], there are lemmas asserting that for all A, $\texttt{Unit} \to A \simeq A$ and $\texttt{Bool} \simeq (\texttt{Bool} \simeq \texttt{Bool})$[3]. So, using univalence to obtain equalities from these equivalences, we obtain after elimination of the equalities:

$$\texttt{map_in_R}\ f\ g : \texttt{map}\ f = g \to \texttt{Bool} \to \texttt{Bool}$$

$$\texttt{R_in_map}\ f\ g : (\texttt{Bool} \to \texttt{Bool}) \to \texttt{map}\ f = g$$

Another lemma that can be proved is that:

$$(g = g) \simeq (\texttt{Bool} \simeq \texttt{Bool})$$

This is true because g is definitionally equal to $\lambda x.\ \texttt{Bool}$.

So, thanks to the univalence axiom, we obtain: $(g = g) = (\texttt{Bool} \simeq \texttt{Bool})$

Additionnaly, $\texttt{R_in_map}\ f\ g$ id gives $\texttt{map}\ f = g$ which we can reinject to prove:

$$\texttt{map_in_R}\ f\ g : \texttt{Bool} \to (\texttt{Bool} \to \texttt{Bool})$$

$$\texttt{R_in_map}\ f\ g : (\texttt{Bool} \to \texttt{Bool}) \to \texttt{Bool}$$

`R_in_mapK` also informs us that `R_in_map` is injective. This implies an inegality between the cardinals:

$$|\texttt{Bool} \to \texttt{Bool}| \leq |\texttt{Bool}|$$

Which, in turns, gives: $4 \leq 2$. This is absurd.

My formalization of this proof can be found in `Optimality.v` [↪ theorem `D4_arrow_left_is_gt3`].

I did not investigate whether the use of axioms was optimal in Table 1, although we strongly suspect it is. This could be an interesting a future development.

[3] There are two automorphisms of `Bool`: the identity and the negation function.

III.2. Attempt at adding new levels to Trocq's hierarchy

III.2.1. Two new levels

Trocq features 6 elements in its lattice of levels: $\{0, 1, 2a, 2b, 3, 4\}$ (see Figure 3). Making this lattice richer could prove useful to represent finer relations between types.

Quentin Vermande, PhD student of my internship advisor, suggested the introduction of two new levels to the lattice: $1a$ and $1b$. The previous level "1" would be renamed to "$1c$".

In the existing hierarchy, level 1 introduces a `map`. Then, levels $2a$ and $2b$ directly state that `map` coincides, one way or the other, with R. But we could also say that R is a functional relation, which hints at the existence of a underlying function – `map` – but without being able to obtain the map.

Quentin noticed these levels appeared naturally when he manually performed a transfer for the proof of induction for the natural numbers.

Let $A, B : \square$ and $R : A \to B \to \square$. The record fields associated to these new levels are:

Level $1a$ `is_total`, with type: $\forall a, \ \left|\sum_{b:B} R\ a\ b\right|$

When $t : T$ is a term, $|t| : |T|$ denotes its truncation (section 3.7 of [20]). The truncation of types is an important concept in HoTT that we unfortunately do not have the space to explain in detail here. The intuition is that if T is a type, $|T|$ is a type inhabited if and only if T is. However, in general, we cannot eliminate $\bar{t} : |T|$ into an inhabitant of T. It does however not mean that given $\bar{t} : |T|$ we cannot extract information about T and eventually recover a $t : T$ for very specific types T [22].

Here, we use type truncation to prevent being able to extract a b out of a term of type `is_total` a. In "standard" Rocq, we would just use $\forall a, \ \exists b, R\ a\ b$ since $\exists x, P\ x$ is in `Prop`.

`is_total` states that when viewed as a function, R gives at least an image for all inputs.

The level $1a$ can be understood as a weaker version of level $2a$, hence its name. Recall the record field associated to level $2a$:

$$\texttt{map_in_R} : \Pi_{(a:\ A)(b:\ B)}(\texttt{map}\ a = b) \to R\ a\ b$$

Contrary to level $1a$, level $2a$ (along with `map`) can be used to obtain b such that $R\ a\ b$. This justifies the use of type truncation to keep $1a$ and $2a$ as separate levels.

Level $1b$ `is_right_unique`, with type: $\forall a\ b\ c, \ R\ a\ b \to R\ a\ c \to b = c$

As its name implies, `is_right_unique` states that when viewed as a function, R has a unique image.

Akin to the relation between $1a/2a$, level $1b$ can be seen as a weaker version of $2b$, hence its level name. Level $2b$'s record field is:

$$\texttt{R_in_map} : \Pi_{(a:\ A)(b:\ B)}(R\ a\ b) \to \texttt{map}\ a = b$$

These two levels insert into the hierarchy as follows:

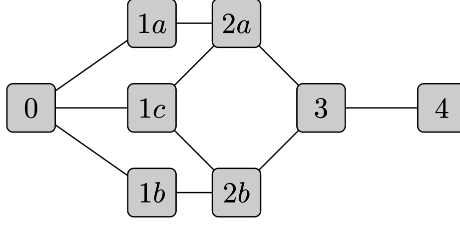


Figure 6: The new lattice of levels

To implement these new levels into Trocq, we need to construct terms $p_{\Pi}^{(m,0)}$, $p_{\to}^{(m,0)}$ when $m \in \{1a, 1b\}$ and study $D_{\Pi}((m,0))$, $D_{\to}((m,0))$. My branch [↪ `1a1b_levels`] contains a Rocq development where I studied the case of the arrow type. The resulting constraints are presented in Table 2. These constraints are proven optimal excepted for $D_{\to}(1a,0)_2$ and $D_{\to}(1b,0)_1$ (orange cells in the table), where it's left to prove that $(1a,0)/(0,1a)$ cannot be achieved. In these cases, optimality is difficult to prove because types T such that it's possible to obtain an inhabitant of T from a term of type $|T|$ are quite restricted even if well understood [22, 23].

m	$D_{\to}(m,0)_1$	$D_{\to}(m,0)_2$
0	$(0,0)$	$(0,0)$
$1a$	$(0,2b)$	$(2a,0)$
$1b$	$(0,2a)$	$(1b,0)$
$1c$	$(0,1c)$	$(1c,0)$
$2a$	$(0,2b)$	$(2a,0)$
$2b$	$(0,2a)$	$(2b,0)$
3	$(0,3)$	$(3,0)$
4	$(0,4)$	$(4,0)$

Table 2: Constraints for the arrow type after introducing the new 1a and 1b levels

Assuming optimality, $D_{\to}(1a)$ is specially worrisome: it has the same constraints as $D_{\to}(2a)$. So, if we're looking for a parametric relation at level $1a$ for an arrow, we can always obtain a relation at level $2a$! This suggests that this level $1a$ is not stable and thus will not be very useful in practice.

This is very unfortunate. We tried several other definitions based on alternate notions of anonymous existence [22], or in the general setting of modalities [20] without success.

III.3. Abstracting over equality types

Trocq relies on axioms in some situations. These axioms are:

- **functional extensionality**: given two types $A : \square$, $B : A \to \square$, given two dependent functions $f, g : \Pi_{a:A}(B\ a)$, functional extensionality states that:

$$(f = g) \simeq (f \sim g)$$

  where $\sim$ is point-wise equality between functions.
- **univalence**: given two types $A, B : \square$, univalence states that:

$$(A = B) \simeq (A \simeq B)$$

Remark: univalence implies functional extensionality.

These axioms are needed to prove equality between product types or between types. The need to prove such equalities stems from the fact that we use equality to state the properties of the hierarchy of parametric relations.

$$\mathtt{Map4}(R) \triangleq \sum \begin{cases} \mathtt{map} : A \to A' \\ \mathtt{map_in_R} : \Pi_{(a:\ A)(b:\ A')} (\mathtt{map}\ a = b) \to R\ a\ b \\ \mathtt{R_in_map} : \Pi_{(a:\ A)(b:\ A')} (R\ a\ b) \to \mathtt{map}\ a = b \\ \mathtt{R_in_mapK} : \Pi_{\substack{a:\ A \\ b:\ A'}} (\mathtt{map_in_R}\ a\ b) \circ (\mathtt{R_in_map}\ \ b\ a) \sim \mathrm{id} \end{cases}$$

Listing 3: Field of the parametricity record at highest level, with uses of equality highlighted

If we instead were to directly replace these equalities with point-wise equality for functions or with equivalence for types, we could spare having to apply the axioms and stop relying on them.

However, it's not as simple as it may seem, because the relation to use as a replacement for equality is specific to the type it's used for. An idea is to attach the relation to the type with a record as such:

```
Record Quiver := {
  Q :> Type;
  hom : Q -> Q -> Type;
}.
```

`hom` $: Q \to Q \to \square$ is the relation over the type Q. With `Quiver`, we would thus require A, A' of type `Quiver` to define $\mathtt{Map4}(R)$, with $R : A \to A' \to \square$.

Of course, we cannot hope to use arbitrary relations as `hom`. The equality type enjoys many properties needed in the definitions of p^{γ}_{Π}, for instance. There properties are: reflexivity, symmetry, transitivity. These properties can be added to the record to create a `Setoid` structure. Let us state its symmetry field:

```
  symmetry: forall x y, (hom x y) -> (hom y x);
```

Pushing the development further revealed another issue. It is important to have structure on elements of `hom x y` as otherwise, we quickly get stuck trying to prove higher-order statements.

Equality in Homotopy Type Theory works well in that regard. For instance, we have:

$$\forall A : \square, \forall (a\ a' : A), \forall (e : a \underset{A}{=} a'),\quad \mathtt{symmetry}\ (\mathtt{symmetry}\ e) = e$$

where $\mathtt{symmetry} : a \underset{A}{=} a' \to a' \underset{A}{=} a$. Equality between equality types also enjoys similar properties: we have a symmetry function the type $\underset{a \underset{A}{=} a'}{=}$, which is itself involutive, and so on. Actually, a very important part of HoTT consists in the study of these higher order structures.

How could we introduce these higher order structures? hom can be taken to be of type $Q \to Q \to V$, with V a type which can itself, for instance, be a `Quiver` or a `Setoid`. Moreover, V often must be a "concrete" type, that it to say, V is equipped with a function $\mathtt{el} : V \to \square$. From $v : V$, `el` can be used to "quantify" over elements of v, by instead quantifying on elements of $\mathtt{el}\ v$.

During the development, many structures naturally appear. An observation we made is that what we're converging to actually closely ressembles enriched category theory. It goes as follows: for a structure based on `Quiver`, the underlying type is the type of points. hom x y is the type of morphisms from x to y. This justifies the use of the terms "`Quiver`" and "hom".

This development coincided with the interests of some researchers at ENS Lyon, who were trying to formalize enriched category theory. After several meetings, Damien Pous, Cyril Cohen, Samuel Arsac and Russ Harmer managed to develop a hierarchy of structures for enriched category theory [24]. We were hoping to draw inspiration from this hierarchy for our usecase in Trocq.

Developping a hierarchy of structures useful for formal proofs is in reality really difficult, both for theoretical issues as well as practical ones – for instance, it involves a lot of boilerplate code. Hierarchy Builder [25] is a Rocq library created to facilitate the creation of a hierarchy of structures. Unfortu-

nately it does, at the moment, not support polymorphic universes, which would be needed in Trocq. Thus, I was not able to directly use it and would have had to reimplement everything that Hierarchy Builder automated.

Combined with some technical difficulties with regards to the tooling available, my lack of intuitions about enriched categories made it difficult for me to work in autonomy, severely hindering my ability to make progress. This is why we decided to stop the investigations here. Even though this exploratory work did not lead to concrete implementation of Trocq replacing equalities with other relations, we believe it is a good first step towards this objective.

Another possible continuation of this work would be to see if, in some cases, we could require less properties on the relation replacing equalities, akin to what Trocq managed to do with the parametricity relation. The use of a hierarchy of relations based on enriched category theory gives hope in that regard since it contains such a hierarchy: from `Quivers` to `Setoids`.

III.4. Maintaining Trocq

During my internship, I also spent time maintaining Trocq. In this subsection, I want to focus on three specific instances of improvements:

- upgrading Trocq from Coq 8.17 to Coq 8.20/Rocq 9.0;
- sharing code between the two Trocq variants, one based on Rocq's Stdlib, the other based on Coq-HoTT;
- unifying Trocq's sort management.

III.4.1. Upgrading Trocq's supported Rocq version

When I started my internship, Trocq supported Coq version 8.17, while the most recent version was Coq 8.20/Rocq 9.0. Trocq aims at being a useful library, so keeping it up-to-date is important. So, that's what I did.

III.4.2. Supporting multiple preludes

Trocq had a "master" branch based on HoTT, and a "prop" branch based on Rocq's stdlib, with additional support for the `Prop` sort.

This lead to duplicate code that required more work to maintain. So, I worked on merging the two branches and deduplicating code. In code base following my changes, Trocq has a directory structure composed of three folders: `std/`, for code specific to the Stdlib variant, `hott/`, for code specific to the HoTT variant and `generic/` for shared Rocq code between the two variants. Since HoTT and Rocq's Stdlib are very different, both variants define a common interface in `Stdlib.v`. For the HoTT variant, this file only includes Coq-HoTT. For the `std` variant, a minimal feature set of Coq-HoTT has been re-implemented.

Because Coq-HoTT is incompatible with Rocq's prelude, a command-line flag must be used to prevent Rocq from automatically loading it. So, both `std/` and `hott/` contain their own `_CoqProject` file. Additionally, the compiled object files of each variant cannot be shared with the other variant, as Rocq's object files record the library context they use. For this reason, all Rocq files found in `generic/` are symbolically linked in `hott/generic/` and `std/generic/`.

Finally, as a general good practice, all Elpi code found in `.v` files was moved to separate files in `elpi/`.

As far as we know, Trocq became the first library supporting different preludes.

III.4.3. Unifying Trocq's sort management

Due to now having multiple variants, Trocq must conditionally support multiple sorts: `Type` for the HoTT variant, `Type` and `Prop` for the Stdlib variant.

Prior to my work on Trocq, code was duplicated for each sort. I unified how sorts were managed in a pull-request [↪ PR #52] (currently pending for review). My changes introduce a generic API for registering sorts to Trocq. Each Trocq variant contains a file named `Param_sort.v`, which calls these APIs.

IV. Conclusion

During this internship, I worked on Trocq, a Rocq library to perform proof transfers modularly. Trocq is useful when we proved a result over a type A, and we wish to obtain this result for a related type B without having to prove it all over again.

Trocq defines a rich hierarchy of relations between types by decomposing type equivalence. Its users are expected to prove the relatedness of two objects by defining a relation of Trocq's hierarchy between them.

Trocq provides a tactic called `trocq` to perform a proof transfer. This tactic implements a translation inspired from the standard univalent parametricity, which it uses, along with additional annotation of types and constraints solving, to transfer the proof.

My work on Trocq was focused both on Trocq's theory and implementation. My various contributions were released as part of Trocq 0.2.0 [13].

On the implementation side, I updated Rocq's version, merged different branches to unify the different Trocq variants (one based on HoTT, the other on Rocq's Stdlib) as well reducing code duplication to improve the maintainability of the code base. As of the time of writing this report, Trocq became available for widespread use in Rocq developments. As far as we know, Trocq is also the first Rocq library to support multiple foundational library: Coq-HoTT and Stdlib from Rocq.

Now, focusing on Trocq's theory. I proved the optimality of the constraints on the input relations for the arrow and forall types. This theoretical question was left open by Trocq's authors. Knowing the constraints optimality gives further confidence in Trocq's hierarchy.

Moreover, I investigated whether Trocq's hierarchy could be extended by adding two new levels. These levels were thought to be an interesting addition which stems from the decomposition of the notion of functional relations. Unfortunately, it turns out the resulting hierarchy is not very useful due to the lack of stability of these new levels when composing relations.

Another important and interesting direction I focused on is the replacement of equalities by relations. Trocq's parametricity hierarchy makes use of the equality type to state, well, equalities between objects. These uses of egality makes Trocq dependent on the functional extensionality and univalence axioms. Moreover, equality might sometimes be too strong of a requirement when "comparing" objects of the same type. I investigated replacing the use of equality by other, user-provided, relations. It lead to theorizing that enriched category theory that a good notion to base Trocq's hierarchy of relations on.

Below is a list of possible future work that would consist in an interesting continuation of my internship.

- continuing to work on the introduction of relations as a replacement of equality, as this should remove dependencies on axioms of Trocq.

  Moreover, this should also allow uses of Trocq in even more situations by weakening the properties needed on the relations replacing equality, as inspired by Trocq's success in doing so for the hierarchy of parametricity levels.

  Waiting for the addition of support of polymorphic universes in Hierarchy Builder is probably wise.
- as a proof-of-concept, it would be interesting to replace CoqEAL refinement logic by Trocq.

- another proof-of-concept use of Trocq could consist in proving the categoricity of math-comp's reals from the categoricity proof of the fourcolor development. This could be an interesting way to showcase Trocq to the math-comp community, extending math-comp with an important theoretical result and testing Trocq's capabilities.

Bibliography

1. Jean Yang: Interview with Xavier Leroy, https://www.cs.cmu.edu/~popl-interviews/leroy.html, last accessed 2025/06/21.
2. The Rocq Development Team: The Rocq Prover, (2025). https://doi.org/10.5281/zenodo.15149629.
3. Thierry Coquand, Gérard P. Huet: The Calculus of Constructions. Inf. Comput. 76, 95–120 (1988). https://doi.org/10.1016/0890-5401(88)90005-3.
4. Thierry Coquand, Christine Paulin: Inductively defined types. Lecture Notes in Computer Science. 417, 50–66 (1988). https://doi.org/10.1007/3-540-52335-9_47.
5. Georges Gonthier: Formal proof - the four-color theorem. Notices of the AMS. 55, 1382–1393 (2008).
6. Georges Gonthier: A computer-checked proof of the Four Color Theorem, https://inria.hal.science/hal-04034866v1, (2023).
7. Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, Laurent Théry: A Machine-Checked Proof of the Odd Order Theorem, (2013). https://doi.org/10.1007/978-3-642-39634-2_14.
8. Xavier Leroy: Formal verification of a realistic compiler. Commun. ACM. 52, 107–115 (2009). https://doi.org/10.1145/1538788.1538814.
9. David Delahaye: A Tactic Language for the System Coq. Lecture Notes in Computer Science. 1955, 85–95 (2000). https://doi.org/10.1007/3-540-44404-1_7.
10. Georges Gonthier, Assia Mahboubi: An introduction to small scale reflection in Coq. Journal of Formalized Reasoning. 3, 95–152 (2010). https://doi.org/10.6092/ISSN.1972-5787/1979.
11. Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, Enrico Tassi: ELPI: Fast, Embeddable, λ-Prolog Interpreter. Lecture Notes in Computer Science. 9450, 460–468 (2015). https://doi.org/10.1007/978-3-662-48899-7_32.
12. Cyril Cohen, Enzo Crance, Assia Mahboubi: Trocq: Proof Transfer for Free, Beyond Equivalence and Univalence. ACM Transactions on Programming Languages and Systems. (2025). https://doi.org/10.1145/3737283.
13. Cyril Cohen, Enzo Crance, Lucie Lahaye, Assia Mahboubi: Trocq - version 0.2.0, (2025). https://doi.org/10.5281/zenodo.15746672.
14. Jean-Philippe Bernardy, Patrik Jansson, Ross Paterson: Proofs for free - Parametricity for dependent types. J. Funct. Program. 22, 107–152 (2012). https://doi.org/10.1017/S0956796812000056.
15. Jean-Philippe Bernardy, Marc Lasson: Realizability and Parametricity in Pure Type Systems. Foundations of Software Science and Computational Structures. 108–122 (2011). https://doi.org/10.1007/978-3-642-19805-2_8.
16. Chantal Keller, Marc Lasson.: Parametricity in an Impredicative Sort. Computer Science Logic. 16, 381–395 (2012). https://doi.org/10.4230/LIPIcs.CSL.2012.381.

17. Nicolas Tabareau, Éric Tanter, Matthieu Sozeau: The Marriage of Univalence and Parametricity. Journal of the ACM. 68, 1–44 (2021). https://doi.org/10.1145/3429979.
18. Christine Paulin-Mohring: Introduction to the Calculus of Inductive Constructions. All about Proofs, Proofs for All. 55, (2015).
19. Maxime Dènès, Anders Mörtberg, Vincent Siles: A Refinement-Based Approach to Computational Algebra in Coq. Lecture Notes in Computer Science. 7406, 83–98 (2012). https://doi.org/10.1007/978-3-642-32347-8_7.
20. The Univalent Foundations Program: Homotopy Type Theory: Univalent Foundations of Mathematics. Institute for Advanced Study (2013).
21. Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau, Bas Spitters: The HoTT library: a formalization of homotopy type theory in Coq, (2017). https://doi.org/10.1145/3018610.3018615.
22. Nicolai Kraus, Martín Escardó, Thierry Coquand, Thorsten Altenkirch: Notions of Anonymous Existence in Martin-Löf Type Theory. Log. Methods Comput. Sci. 13, (2017). https://doi.org/10.23638/LMCS-13(1:15)2017.
23. Nicolai Kraus: The Truncation Map $|_| : \mathbb{N} \to \|\mathbb{N}\|$ is nearly Invertible, https://homotopytypetheory.org/2013/10/28/the-truncation-map-_-%E2%84%95-%E2%80%96%E2%84%95%E2%80%96-is-nearly-invertible/, last accessed 2025/06/21.
24. Damien Pous, Cyril Cohen, Samuel Arsac, Russ Harmer: A development of a hierarchy of structures in Rocq for enriched categories, https://github.com/damien-pous/cats/.
25. Cyril Cohen, Pierre Roux, Kazuhiko Sakaguchi, Enrico Tassi: Hierarchy Builder: Algebraic hierarchies made easy in Coq with Elpi. LIPIcs. 167, 34:1–34:21 (2020). https://doi.org/10.4230/LIPICS.FSCD.2020.34.