Trocq parametricity translations for inductive types

Tomás Vallejos Parada
Inria
Nantes, France
tomas.vallejos@inria.fr

Abstract

This submission describes an extension of the Trocq plugin for proof transfer. It automates the generation of the lattice of parametricity translations from the Trocq hierarchy for a significant class of inductive types, thus removing this burden from the user.

Keywords

Proof transfer, Metaprogramming, Parametricity translation, Rocq prover

1 Introduction

Even in major interactive theorem provers, the bureaucracy of proof transfer between different representations of the same concept remains one of the main sources of frustrating bureaucracy in formal proofs. For instance, applying a lemma about sequences of integers, of type `list Z`, to a sequence of natural numbers, of type `list nat`, may well span several lines in a proof script. Implementations of parametricity translations for the Calculus of Inductive Constructions, such as `paramcoq` [3] or univalent parametricity [6], can be used to automate such mundane proof steps, by assembling automatically the essential building blocks of these transfer proofs. The recent Trocq plugin [2] can actually compute proofs of implications justifying the transfer of a goal from type `list Z` to type `list nat`, provided a proof of the embedding relation from type `nat` into type `Z` and a proof that the polymorphic type `list` propagates this relation pairwise, as expected.

While the former proof probably has to be implemented "by hand", the latter can actually be automated by a systematic process, driven by the syntactic definition of the inductive type `list`. Sadly as of today, users still have to prove all these building blocks by hand to benefit from the univalent parametricity of Trocq. More precisely, for any type T used in a goal, the user has to make sure that a relation $[\![T]\!]$ is known to the plugin, which satisfies the corresponding instance of the abstraction theorem. This work generalizes existing plugins, such as derive [8], for computing the parametricity translation of a substantial class of inductive types, into an extension to the translations involved in the Trocq plugin. This class includes polymorphic recursive types, but not type families.

2 Context

To keep the presentation self-contained we remind some of the Trocq's [2] definitions.

Definition 2.1. For any types $A, B : \square$, a relation $R : A \to B \to \square$, is a *univalent map*, denoted $\mathsf{IsUmap}(R)$ when there exists a function $m : A \to B$ together with:

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

The definition of a univalent map consists of four pieces of data: the maps m, $g1$ and $g2$, and a coherence property between $g1$ and $g2$. The coherence depends on the three formers, while $g1$ and $g2$ depend on the map m. These dependencies enable the description of a hierarchy for which the amount of data indicates how close the relation is to being a univalent map.

Definition 2.2. For $n \in \{0, 1, 2_a, 2_b, 3, 4\}$, each M_i defines a class of maps in the hierarchy follows:[1]

$$\mathsf{M}_0\ R \triangleq .$$
$$\mathsf{M}_1\ R \triangleq (A \to B)$$
$$\mathsf{M}_{2_a}\ R \triangleq \Sigma m : A \to B.\, G_{2_a}\ m\ R \quad \text{with } G_{2_a}\ m\ R \triangleq \Pi a\ b.\, m\ a = b \to R\ a\ b$$
$$\mathsf{M}_{2_b}\ R \triangleq \Sigma m : A \to B.\, G_{2_b}\ m\ R \quad \text{with } G_{2_b}\ m\ R \triangleq \Pi a\ b.\, R\ a\ b \to m\ a = b$$
$$\mathsf{M}_3\ R \triangleq \Sigma m : A \to B.\, (G_{2_a}\ m\ R) \times (G_{2_b}\ m\ R)$$
$$\mathsf{M}_4\ R \triangleq \Sigma m : A \to B.\, \Sigma(g_1 : G_{2_a}\ m\ R).\, \Sigma(g_2 : G_{2_b}\ m\ R).\, G_4\ m\ g_1\ g_2 \quad \text{with } G_4\ m\ g_1\ g_2 \triangleq \Pi a\ b.\, (g_1\ a\ b) \circ (g_2\ a\ b) \doteqdot id$$

Similarly, this hierarchy of maps enables the definition of a hierarchy of relations.

Definition 2.3. For $n, k \in \{0, 1, 2_a, 2_b, 3, 4\}$, and $\alpha = (n, k)$, relation $\boxdot^\alpha : \square \to \square \to \square$, is defined as:

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

where the *map class* $\mathsf{Class}_\alpha\ R$ itself unfolds to a pair type $(\mathsf{M}_n\ R) \times (\mathsf{M}_k\ R^{-1})$, with R^{-1} just swapping the order of arguments for R.

Logical equivalence pertains to $\mathsf{Class}_{(1,1)}$, $\mathsf{Class}_{(4,0)}$ describe univalent maps, $\mathsf{Class}_{(4,2_a)}$ and $\mathsf{Class}_{(4,2_b)}$ characterize univalent maps with an explicit partial left and right inverse, respectively. The relation between *nat* and *Z* alluded in the introduction is an instance of $\mathsf{Class}_{(4,2_b)}$. Last, $\mathsf{Class}_{(4,4)}$ characterizes type equivalences.

We now present the set of inductives for which our algorithm is defined. The algorithm we propose targets recursive inductives with type parameters, that does not use indices, nor nesting, nor non-uniform parameters.

Definition 2.4. For $n, k \in \mathbb{N}$, and for $i \in \{1, ..., k\}$, $c_i \in \mathbb{N}$. We describe an inductive type I, with n type parameters, k constructors

[1] For the sake of readability, we omit implicit arguments, *e.g.*, although M_i has type $\lambda(T_1\ T_2 : \square).\ (T_1 \to T_2 \to \square) \to \square$, we write $\mathsf{M}_n\ R$ for $(\mathsf{M}_n\ A\ B\ R)$.

with c_i arguments each, and every constructor argument $a_{i,j}$ of type $T_{i,j}$ and $j \in \{1, ..., c_i\}$, as follows.

$$I(A_1 : \square, ..., A_n : \square) : \square_{\triangleq}$$
$$|K_1(a_{1,1} : T_{1,1}) ... (a_{1,c_1} : T_{1,c_1})$$
$$...$$
$$|K_k(a_{k,1} : T_{k,1}) ... (a_{k,c_k} : T_{k,c_k})$$

We allow types $T_{i,j}$ to be part of the type parameters, or to refer to a previously defined type. We write $\overline{A}$ to denote $A_1, ..., A_n$ for readability.

3 The algorithm

Our algorithm allows us to inhabit the following type:

$$\overline{\Pi(A\ B : \square)(R : A \to B \to \square)(Ri : \boxdot^{i,0}\ A\ B)}. \qquad (1)$$
$$\boxdot^{i,0}\ \ (I\ \overline{A})\ (I\ \overline{B}) \qquad \text{with } i \in \{0, 1, 2_a, 2_b, 3, 4\}$$

The difficulty comes from the fact that we want to proceed gradually, as opposed to applying weakening. One thus has to design G_{2_a} and G_{2_b} foreseeing that their composition has to cancel out when building the respective M_4. If one eliminates the identity in G_{2_a}, or returns a complex identity in G_{2_b}, the composition gets blocked on reduction. The goal of the algorithm is to avoid that situation. To address this issue we design maps that compute, i.e., maps returning a term whose head is not the result of a transport.

4 A concrete example

For illustration purposes, we show the terms our algorithm automatically generates for the inductive `list`. Their generation requires some boilerplate, thus for some terms we only specify their types. We proceed using a bottom-up approach, that is, we focus on building the terms incrementally, from maps $\text{Class}_{0,0}\ R$ up to $\text{Class}_{4,0}\ R$.

We use the parametricity translation of lists as the relation to be equipped with the univalent map pieces of data.

```
Inductive listR (A B : Type) (AR: A -> B -> Type)
: list A -> list B -> Type :=
| nilR : listR A B AR nil nil
| consR : forall a b aR l l' lR,
      listR A B AR (cons a l) (cons b l')
```

As a map, we use the map for lists, spelled out using Trocq's hierarchy, that is, we use a relation $\boxdot^{(1,0)} A\ B$ instead of a function of type $A \to B$.

```
Fixpoint map_list (A B : Type) (AR : Param10.Rel A B)
  (l: list A) : list B :=
    match l with
    | nil => nil
    | cons a l => cons (map AR a) (map_list _ _ AR l)
    end.
```

With these two definitions we generate the following terms:

```
Fixpoint map_in_R_list (A B : Type)
(AR : Param2a0.Rel A B) (l : list A) (l' : list B) :
 map_list A B AR l = l' -> listR A B AR l l'.
(*...*)
Defined.

Fixpoint R_in_map_list (A B : Type)
(AR : Param2b0.Rel A B) (l : list A) (l' : list B)
(lR : listR A B AR l l') : map_list A B AR l = l'.
(*...*)
Defined.

Fixpoint map_in_RK_list (A B : Type)
(AR : Param40.Rel A B): forall (l1 : list A)
(l2 : list B) (lR: listR A B AR l1 l2),
  map_in_R_list _ _ AR l1 l2
    (R_in_map_list _ _ AR l1 l2 lR) = lR.
(*...*)
Defined.
```

5 Implementation

We implement this algorithm in the Rocq prover [9] using Rocq-Elpi [7], building on top of the DERIVE plugin described in [8]. The parametricity translation is generated using PARAM2, implemented by Cyril Cohen. Our implementation of the map m is a modification of DERIVE's algorithm for generating a map.

Our implementation is available at https://github.com/Tvallejos/trocq/tree/rocqpl-26. The directory `std/algo/elpi/` contains the implementation of the parametricity translation for the hierarchy of Trocq. Examples on how to use it can be found in the `std/algo/tests` directory.

6 Conclusion and related work

Parametricity translations for dependent types [1, 4] allow proofs to be transferred in proof assistants. The work of Tabareau et al. in [5, 6] allows proof to be transfered under type equivalences, albeit using the univalence axiom[10]. Newer frameworks, also exploiting parametricity to perform proof transfer, such as Trocq [2], have unified the transfer across type equivalences and weaker relations, while avoiding the use of univalence where it is not needed.

Our work bridges an automation gap for deriving the parametricity translation for inductives in Trocq's hierarchy, making proof transfer more usable. Up to our knowledge, no similar automation exists for other implementations of parametricity models, including the univalence translation.

References

[1] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. 2010. Parametricity and dependent types. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, Paul Hudak and Stephanie Weirich (Eds.). ACM, 345–356. doi:10.1145/1863543.1863592

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

[3] Chantal Keller and Marc Lasson. 2012. Parametricity in an Impredicative Sort. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France (LIPIcs, Vol. 16)*, Patrick Cégielski and Arnaud Durand (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 381–395. doi:10.4230/LIPICS.CSL.2012.381

[4] 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.

[5] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. 2018. Equivalences for free: univalent parametricity for effective transport. *Proc. ACM Program. Lang.* 2, ICFP (2018), 92:1–92:29. doi:10.1145/3236787

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

[7] Enrico Tassi. 2018. Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λProlog dialect). In *The Fourth International Workshop on Coq for Programming Languages*. Los Angeles (CA), United States. https://inria.hal.science/hal-01637063

[8] Enrico Tassi. 2019. Deriving Proved Equality Tests in Coq-Elpi: Stronger Induction Principles for Containers in Coq. In *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA (LIPIcs, Vol. 141)*, John Harrison, John O'Leary, and Andrew Tolmach (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 29:1–29:18. doi:10.4230/LIPICS.ITP.2019.29

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

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