Artifact Report: Trocq: Proof Transfer for Free, With or Without Univalence

Cyril Cohen, Enzo Crance, Assia Mahboubi

Artifact Report: Trocq: Proof Transfer for Free, With or Without Univalence

Cyril Cohen[1], Enzo Crance[2,3], and Assia Mahboubi[2,4]

[1] Université Côte d'Azur, Inria
Cyril.Cohen@inria.fr
[2] Nantes Université, École Centrale Nantes, CNRS, INRIA, LS2N, UMR 6004
{Enzo.Crance,Assia.Mahboubi}@inria.fr
[3] Mitsubishi Electric R&D Centre Europe
[4] Vrije Universiteit Amsterdam, The Netherlands

Trocq [?] is both the name of a calculus, describing a parametricity framework, and of a Coq plugin [?] that provides tactics for performing representation changes in goals, as well as vernacular commands for specifying the expected translations. More precisely, from an initial goal of type `G`, the `trocq` tactic simultaneously computes using the Trocq calculus [?] a translation `G'` and a justification `w : G' -> G`. If successful, the user is thus left with proving `G'`.

The plugin orchestrates this double synthesis, by assembling existing building blocks known to the tactic, in the course of a linear traversal of the input term `G`. These building blocks are of two natures. First, the actual rules of the parametricity framework [?] govern the synthesis rule attached to each term construction of CC_ω. The other nature of building blocks is the collection of registered pairs of user-defined constants. These pairs come equipped with a witness of their relatedness at some level, a data registered via the `Trocq Use` command. When the linear traversal of the input term hits a constant, it queries the database registering these user-defined relations, looking for the corresponding constant and witness to be used in the synthesis.

The Trocq plugin is implemented in Elpi [?]: a dialect of λProlog which can be used as a meta-language for Coq, through the Coq-Elpi [?] plugin. The latter encodes Coq terms in higher-order abstract syntax (HOAS) which provides native support for bound variables, complemented by a comprehensive API (typechecking, elaboration, interacting with the global environment, etc).

1 Example

Let us translate the induction principle associated with type `nat`, the unary representation of ℕ, to type `N`, the binary one. Types `nat` and `N` are equivalent and we use the `Trocq Use` command to register such pairs of related types:

```
Definition RN : (N <=> nat)%P := Iso.toParamSym N.of_nat_iso.
Trocq Use RN. (* registering a pair of related types *)
```

Proof `RN` coerces to a relation of type `N -> nat -> Type`, and we also register proofs that it relates the respective zero and successor constants of these types:

```
Definition RN0 : RN 0%N 0%nat. Proof. done. Qed.
Definition RNS m n : RN m n -> RN (N.succ m) (S n). Proof. by case. Qed.
Trocq Use RN0. Trocq Use RNS. (* registering related constants *)
```

We can now use tactic `trocq` to prove a useful induction principle on type `N`:

```
Lemma N_Srec : forall (P : N -> Type), P 0%N ->
    (forall n, P n -> P (N.succ n)) -> forall n, P n.
Proof. trocq. (* replaces N by nat in the goal *) exact nat_rect. Qed.
```

Inspecting the proof term actually reveals that univalence was not needed in the proof of `N_Srec`. The `example` directory of the artifact provides more examples, for weaker relations than equivalences, and beyond representation independence.

2 Architecture of the plugin

A Trocq parametricity sequent $\Delta \vdash_t M \mathbin{@} A \sim M' \mathbin{\because} M_R$ expresses that *terms M and M' are related at type A with witness M_R* in context Δ. Unlike standard, unequivocal parametricity translations, each construct of CC_ω gives rise to a *family* of possible synthesis rules, indexed by annotations on M and A.

Encoding CC_ω^+ . To implement the annotation calculus CC_ω^+ , we just annotate Coq's sort `Type` with a pair (n, m) using *convertible* synonyms `(PType n m)`, where `PType := fun (_ _ : label) => Type`. The two thrown-away arguments code for the annotation. In the course of the synthesis, arguments of certain occurrences of `PType` are left as holes and filled by a constraint solving algorithm.

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

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

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

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

Handling of constants. Finally, the traversal of the input term collects *constraints* on the annotations, as multiple valid solutions might exists: for instance, an implication might be obtained from weakening an equivalence. The algorithm strives to minimize the requirements on the user-defined building blocks, which also amounts to minimizing the dependency on axioms. This inference procedure is formalized as a finite domain constraint solving problem, and implemented using *Constraint Handling Rules* (CHR) language [**?**], as available in Elpi.

3 Related work

In the context of type theory, Barthe and Pons [**?**] already noticed that the computational content of type isomorphisms can serve proof transfer. The first implementation report of a tool based on this idea appeared soon after [**?**]. Implemented in a meta-language and based on proof rewriting, this heuristic translation produced a candidate proof term from an existing proof term, with no formal guarantee, not even that of being well-typed. Generalized rewriting [**?**], 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 more general instances of proof transfer, *e.g.*, , datatype representation change and quasi-PERs (QPER, or zig-zag complete relations) [**?**], essentially heterogeneous.

The other proof transfer methods we are aware of all address the case of heterogeneous relations. Incidentally, they can thus also be used for the homogeneous case, and thus for generalized rewriting, although this special case is seldom emphasized. The Coq Effective Algebra Library (CoqEAL) [**?**,**?**] and the Isabelle/HOL transfer package [**?**,**?**,**?**,**?**], 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 [**?**], these tools address the case of a transfer between a subtype of a certain type A and a quotient of a certain type B, *i.e.*, the case of a trivial QPER in which the zig-zag morphism is a surjection from A to B.

Modern approaches to proof transfer rely on univalence, either as an axiom, in the case of univalent parametricity [**?**] or as a computing primitive [**?**]. Key ingredients of univalent parametricity were already present in earlier seemingly unpublished work [**?**], implemented using an ancestor of the MetaCoq library [**?**].

The columns of Table **??** lists these tools in chronological order, and indicates when the features listed as lines are available (✓), not available (✗) or only partially available (✎). Transfer along *heterogeneous relations*, and while the oldest tool operates via a monolithic translation of an input proof term, others rather prove an *internal* implication lemma. *Anticipation* [**?**] refers to the need to define a dedicated structure for the signature to be transported. *Binders* (∀) can prevent transfer, as well as *dependent types*, which require univalence.

	[?]	[?]	[?]	[?]	[?]	[?]	[?]	[?]	Trocq
Heterogen. rel.	✓	✗	✓	✓	✓	✓	✓	✓	✓
Internal	✗	✓	✓	✓	✓	✓	✓	✓	✓
No anticipation	✓	✓	✓	✓	✓	✓	✗	✓	✓
Under ∀	✓	✓	✗	✓	✓	✓	✓	✓	✓
Dep. types	✓	✗	✗	✗	✗	✓	✓	✗	✓
Univalence-free	✓	✓	✓	✓	✓	✗	✗	✓	✓
Subrelations	✗	✓	✗	✗	✗	✗	✗	✗	✎
QERs	✗	✎	✎	✎	✎	✗	✓	✗	✎
Subtyping	✗	✗	✎	✎	✎	✗	✗	✎	✎
	Coq	Coq	Coq	Isabelle/HOL	Coq	HoTT	CubicalAgda	Coq	Coq or HoTT

Table 1. Comparison of proof transfer automation devices

Acknowledgments. The authors would like to thank Enrico Tassi and the anonymous reviewers. This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101001995).