Automatic and Transparent Transfer of Theorems along Isomorphisms in the Coq Proof Assistant

Théo Zimmermann, Hugo Herbelin

Automatic and Transparent Transfer of Theorems along Isomorphisms in the Coq Proof Assistant

Theo Zimmermann[1] and Hugo Herbelin[2]

[1] École Normale Supérieure, Paris, France
theo.zimmermann@ens.fr
[2] Inria Paris-Rocquencourt, Paris, France
hugo.herbelin@inria.fr

Abstract. In mathematics, it is common practice to have several constructions for the same objects. The mathematicians will identify them modulo isomorphism and will not worry later on which construction they use, as theorems proved for one will be valid for all.
When working with proof assistants, it is also common to see several data-types representing the same objects. This work aims at making the use of several isomorphic constructions as simple and as transparent as it can be done informally in math. This requires inferring automatically the missing proof-steps.
We are conceiving an algorithm to do so and implementing it as a plugin for Coq [3].

1 Introduction

Linear maps being identified to matrices, real numbers being seen once as equivalence classes of Cauchy sequences, once as Dedekind cuts, once as infinite sequences of digits, once as a subset of complex numbers... There is a great many cases when identifying several constructions of the same objects can be useful in mathematics. In particular, proofs are then done on the most convenient one but theorems apply to all.

In formal systems like Coq [4], a canonical example is the various constructions available for natural numbers. The most natural construction and the closest to the mathematical view is unary (`0`, `S 0`, `S (S 0)` and so on) while the more efficient binary construction is closest to what is available in most programming languages.

When several constructions coexist, they often share an axiomatic representation, abstracting away from the internal details. In Coq, it is possible to do proofs directly on the axiomatic representation thanks to the module and functor system [1]. While this has the advantage of factoring proofs, it makes also

[3] This plugin introduces a new tactic called `exact modulo`. Its most recent version is available on the web at https://github.com/Zimmi48/transfer.

the proof harder as it does not allow taking advantage of the specifics of the implementation.

The purpose of this work is to make easy transporting theorems to all isomorphic constructions even when the proof relies on one particular such construction. While in an informal setting, the mathematician will declare that "we can identify the two structures" once she has proved they were isomorphic and will proceed from there, our goal is to justify that claim because it will be that missing justification that the proof checker will ask for. Moreover, we need to determine when this justification is missing and insert it automatically.

Although we focus on isomorphic structures in our description of the problem and in our examples, we want to emphasize that we thrive to be as general as possible and require as little as possible to allow the automatic transfer of a theorem. Sometimes an isomorphism is required but sometimes it is not. Our algorithm will typically allow the following transfer:

Example 1. Take two sets A and B. If we have the following result on the first set:

Axiom 1 (A is empty).

$$\forall x \in A, \bot \ .$$

then a surjective function $f : A \to B$ is all we need to transfer the result and get:

Theorem 1 (B is empty).

$$\forall x \in B, \bot \ .$$

Here is the complete corresponding Coq development (using our plugin – although in that case, it is extremely easy to build the proof by hand):

```
Parameter A B : Set.
Axiom emptyA : ∀ x : A, False.
Parameter f : A → B.
Parameter g : B → A.
Axiom surjf : ∀ x : B, f (g x) = x.
Declare Surjection f by (g, surjf).
Theorem emptyB : ∀ x : B, False.
  exact modulo emptyA.
Qed.
```

In the remainder of this text, we will start by presenting our current algorithm which is able to transfer a limited but already interesting set of theorems. Then, we will detail our ideas to generalize it. Finally, we will compare our approach to previous related works.

2 How to Transfer a Theorem

To start, we are limiting ourselves to transferring first-order formulas containing only universal quantifiers, implication and relations.

We only require from the user to provide a set of surjective functions between related data-types, along with a proof of surjectivity, and transfer lemmas.

That is, we can relate two data-types A and A' by producing a function $f : A \to A'$ and a proof that f is surjective. To ease our task, we will require that the proof that f is surjective be given by producing a right-inverse g and a proof that

$$\forall x \in A', f(g(x)) = x \ .$$

If the user wishes to transfer a relation $R \in A \times A \times \ldots \times A$ to a relation $R' \in A' \times A' \times \ldots \times A'$, she must provide a transfer lemma of the form

$$\forall x_1 \ldots x_n \in A, R(x_1, \ldots, x_n) \Rightarrow R'(f(x_1), \ldots, f(x_n))$$

where f is called the transfer function between R and R'.

The declared surjections and transfer lemmas will be stored in tables (maps). A given surjection can be retrieved by looking for a pair of data-types while a given transfer lemma can be retrieved by looking for a pair of relations. There can be only one stored item for each key which prevents to define several distinct isomorphisms between two structures.

Example 2 shows how this is enough for transferring interesting theorems from one data-type to another.

Example 2. Suppose we are given two data-types to represent $\mathbb{N}$, called nat and N together with two relations $\leq_{\text{nat}}$ and $\leq_{\text{N}}$.

We know nothing of their implementation but we are also given two functions $\text{N.to\_nat} : \text{N} \to \text{nat}$ and $\text{N.of\_nat} : \text{nat} \to \text{N}$ and the four accompanying axioms:

Axiom 2 (Surjectivity of N.to_nat).

$$\forall x \in \text{nat}, \text{N.to\_nat}(\text{N.of\_nat}(x)) = x \ .$$

Axiom 3 (Surjectivity of N.of_nat).

$$\forall x' \in \text{N}, \text{N.of\_nat}(\text{N.to\_nat}(x')) = x' \ .$$

Axiom 4 (Transfer from $\leq_{\text{N}}$ to $\leq_{\text{nat}}$ by N.to_nat).

$$\forall x', y' \in \text{N}, x' \leq_{\text{N}} y' \Rightarrow \text{N.to\_nat}(x') \leq_{\text{nat}} \text{N.to\_nat}(y') \ .$$

Axiom 5 (Transfer from $\leq_{\text{nat}}$ to $\leq_{\text{N}}$ by N.of_nat).

$$\forall x, y \in \text{nat}, x \leq_{\text{nat}} y \Rightarrow \text{N.of\_nat}(x) \leq_{\text{N}} \text{N.of\_nat}(y) \ .$$

Finally, we are given the following result to transfer:

Axiom 6 (Transitivity of $\leq_{\text{nat}}$).

$$\forall x, y, z \in \text{nat}, x \leq_{\text{nat}} y \Rightarrow y \leq_{\text{nat}} z \Rightarrow x \leq_{\text{nat}} z \ .$$

All these results enable us indeed to transfer Axiom 6 into Theorem 6.

Theorem 6 (Transitivity of $\leq_{\mathrm{N}}$).

$$\forall x', y', z' \in \mathrm{N}, x' \leq_{\mathrm{N}} y' \Rightarrow y' \leq_{\mathrm{N}} z' \Rightarrow x' \leq_{\mathrm{N}} z' \ .$$

Proof. Let $x', y', z' \in \mathrm{N}$ and assume that the two following hold:

$$x' \leq_{\mathrm{N}} y' \ , \tag{1}$$

$$y' \leq_{\mathrm{N}} z' \ . \tag{2}$$

From (1) (respectively (2)) and Axiom 4, we draw

$$\mathrm{N.to\_nat}(x') \leq_{\mathrm{nat}} \mathrm{N.to\_nat}(y') \ , \tag{3}$$

$$\mathrm{N.to\_nat}(y') \leq_{\mathrm{nat}} \mathrm{N.to\_nat}(z') \ . \tag{4}$$

We can now apply Axiom 6 to $\mathrm{N.to\_nat}(x')$, $\mathrm{N.to\_nat}(y')$ and $\mathrm{N.to\_nat}(z')$ and conclude

$$\mathrm{N.to\_nat}(x') \leq_{\mathrm{nat}} \mathrm{N.to\_nat}(z') \ . \tag{5}$$

We then apply Axiom 5 to get

$$\mathrm{N.of\_nat}(\mathrm{N.to\_nat}(x')) \leq_{\mathrm{N}} \mathrm{N.of\_nat}(\mathrm{N.to\_nat}(z')) \ . \tag{6}$$

That is (rewriting with Axiom 3):

$$x' \leq_{\mathrm{N}} z' \ . \tag{7}$$

□

You will have noticed that Axiom 2 has not been useful here. It would have been if there had been a quantification to transfer inside one of the hypotheses.

Algorithm 1 takes as input two formulas (called *theorem* and *goal*) differing only in the data-types that are quantified over and in the relations they contain, as well as a proof of *theorem*. It outputs a proof of *goal* provided that the differences between the two formulas all correspond to previously declared surjections and transfer lemmas.

The algorithm recurses over the structure of the two formulas (which must be the same). There are two main cases: when the formulas are atoms (i.e. in our case, relations applied to arguments) or dependent products. Dependent products are the way in which the Calculus of Constructions [5], the logical base of Coq, models both universal quantification and implication. An implication is just a non-dependent product, i.e. $A \Rightarrow B$ is just an abbreviation for $\forall x : A, B$ when x does not appear in B.

Note also that in the Calculus of Constructions as well as in any other type-theory-based logic, proofs can be viewed as programs, and in particular the proof $\rho_{A \Rightarrow B}$ of an implication $A \Rightarrow B$ can be viewed as a function that takes a proof ρ_A of A as argument and produces a proof $\rho_{A \Rightarrow B}(\rho_A)$ of B.

Algorithm 1 Transfer a Theorem

Precondition: In the environment Γ, F and F' are two well-defined formulas and ρ_F is a proof of F.

Postcondition: ExactModulo(Γ, F, F', ρ_F) is a proof of F' in environment Γ or it is a failure.

function ExactModulo(Γ, F, F', ρ_F)
  if $F = F'$ **then**
    return ρ_F
  else if $F = R(x_1, \ldots, x_n) \wedge F' = R'(x'_1, \ldots, x'_n)$ **then**
    $f \leftarrow$ transfer function between R and R'
      ▷ **return** failure if it does not exist
    $\rho_{\text{transfer}} \leftarrow$ proof of compatibility of f with respect to R and R'
    for $i \leftarrow 1$ to n **do**
      if $x'_i \neq f(x_i)$ **then**
        return failure
    return $\rho_{\text{transfer}}(x_1, \ldots, x_n, \rho_F)$
  else if $F = \forall x : A, B \wedge F' = \forall x' : A', B'$ **then**
    $\Gamma \leftarrow \Gamma, x' : A'$
    $x \leftarrow$ ExactModulo(Γ, A', A, x')
    if $x \neq$ failure **then**
      $\rho_{\text{rec}} \leftarrow$ ExactModulo$(\Gamma, B, B', \rho_F(x))$
        ▷ **return** failure if $\rho_{\text{rec}} =$ failure
      return $\lambda x'.\, \rho_{\text{rec}}$
    else
      $f \leftarrow$ surjection from A to A' ▷ **return** failure if it does not exist
      $g \leftarrow$ right-inverse of f
      $\rho_{\text{surjection}} \leftarrow$ proof that g is a right-inverse of f
      $B_{\text{subst}} \leftarrow B$ where x was replaced by $g(x')$
      $B'_{\text{subst}} \leftarrow B'$ where x' was replaced by $f(g(x'))$ in covariant places
      $\rho_{\text{rec}} \leftarrow$ ExactModulo$(\Gamma, B_{\text{subst}}, B'_{\text{subst}}, \rho_F(g(x')))$
        ▷ **return** failure if $\rho_{\text{rec}} =$ failure
      Now $\lambda x'.\, \rho_{\text{rec}}$ is a proof of $\forall x' : A', B'_{\text{subst}}$. With the help of $\rho_{\text{surjection}}$ we can transform it into $\rho_{F'}$ a proof of $\forall x' : A', B'$.
      return $\rho_{F'}$
  else
    return failure

You will have noticed the strange choice of substituting x' with $f(g(x'))$ only in covariant places. As $x' = f(g(x'))$, we could have done the substitution wherever we liked. We do it only in covariant places so that the formulas in the recursive calls will have exactly the right form when reaching the atomic case (relations). One can convince oneself that substituting in covariant places is enough by observing what it gives on the last example (transitivity of $\leq_{\text{N}}$) while remembering that the right-hand side of an implication is covariant while the left-hand side is contravariant.

We could then have added support for logical connectives such as $\wedge$ and $\vee$ or the existential quantifier $\exists$ but as they play no specific role in the Calculus of Constructions (unlike universal quantification and implication), we rather want a more general way of treating any such addition. As for the negation $\neg A$, in Coq it is defined as $A \Rightarrow \bot$ so it is already supported provided we unfold its definition first.

3 Generalizing

There are a lot of things that Algorithm 1 does not handle yet but would be nice to have.

Functions. So far we have considered only relations. Even though any function can be expressed as a relation, it would be a lot more convenient to be able to transfer functions directly. Given that relations are represented as functions to the special sort `Prop` in Coq, what we need is a generalization where functions to any type would be supported, as well as internal operators.

New connectives. We want to be able to handle logical connectives such as $\wedge$ and $\vee$ but also various other combinators and non-propositional functions. For instance, we should be able to transfer theorems involving equality.

Other equivalence relations. Currently, Leibniz (structural) equality plays a special role as it has to appear in the surjection lemmas. Leibniz equality has the advantage of allowing rewriting in any subterm. But techniques have already been devised [6] to allow rewriting with other equivalence relations and we plan to inspire from them.

No right-inverse. For simplicity, we have asked so far for proofs of surjectivity which involved producing a right-inverse. This has a major drawback. Indeed, surjectivity is equivalent to having a right-inverse only if we admit the Axiom of Choice. We want our algorithm to be as general as possible, therefore we will work to remove that requirement.

3.1 Generalizing Declarations

Transfer lemmas. The Coq Morphisms library[4] introduces a new notion of respectful morphisms for a binary homogeneous relation. We draw from [2] the idea of using the generalized heterogeneous version for our transfer declarations. Heterogeneous relations bring us the ability to relate objects from one data-type with objects from another data-type.

[4] The Coq Morphisms library is part of the work of Matthieu Sozeau [6] to generalize rewriting for equivalence relations that are not Leibniz equality. Its documentation is available online at https://coq.inria.fr/library/Coq.Classes.Morphisms.html.

We will note

```
(R ##> R') f g := ∀ (x : A) (x' : B), R x x' → R' (f x) (g x') .
```

This can also be seen as a (commutative) diagram.

$$\begin{array}{ccc} A & \xleftrightarrow{R} & B \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle g} \\ C & \xleftrightarrow{R'} & D \end{array}$$

Then, to continue on our previous example,

Example 3. let us consider an heterogeneous binary relation `natN` relating elements of `nat` with elements of `N`. One possible definition would be:

```
Definition natN x x' := N.of_nat x = x'.
```

Then, we can declare how to transfer various functions and relations:

```
Theorem le_transfer : (natN ##> natN ##> impl) le N.le.
```

where `le` represents $\leq_{\mathrm{nat}}$, `N.le` represents $\leq_{\mathrm{N}}$ and `impl` is a relation corresponding to the implication (also, note that `##>` is right-associative). That is, after unfolding the definitions of `natN`, `##>` and `impl`:

```
Theorem le_transfer :
    ∀ (x : nat) (x' : N), N.of_nat x = x' →
    ∀ (y : nat) (y' : N), N.of_nat y = y' →
    le x y → N.le x' y'.
```

Considering two new Boolean functions `iszero_nat` and `iszero_N`, we can make explicit how they relate in the following way:

```
Theorem iszero_transfer :
    (natN ##> @eq bool) iszero_nat iszero_N.
```

where `@eq bool` is the Boolean equality.

Finally, considering two operations `Nat.add` and `N.add`:

```
Theorem plus_transf : (natN ##> natN ##> natN) Nat.add N.add.
```

Surjection lemmas. That very same idea of respectful morphisms can be used to replace the surjection declarations we used so far. Just as we had replaced the implication $\rightarrow$ by a new relation `impl`, we will use a new relation `@all` to represent $\forall$:

```
@all A (λ x : A, B) := ∀ x : A, B .
```

Then, to continue our previous example,

Example 4. the surjection lemma from `nat` to `N` can be replaced by:

```
Theorem natN_surj :
    ((natN ##> impl) ##> impl) (@all nat) (@all N).
```

That is, after unfolding the definitions of `natN`, `##>`, `impl` and `@all`:

```
Theorem natN_surj : ∀ P P',
    (∀ (x : nat) (x' : N), N.of_nat x = x' → P x → P' x') →
    (∀ x : nat, P x) → ∀ x' : N, P' x'.
```

In the general case, we have proved[5] that any surjection declaration can be replaced by a declaration of the form

```
((R ##> impl) ##> impl) (@all A) (@all A')
```

but the latter is more general (it does not require a right-inverse for the surjective function and can handle more relations than just Leibniz equality).

3.2 Transfer to the context

In [6], Matthieu Sozeau gives a set of inference rules to find where a rewrite can occur and the proof that the rewrite is correct. Building the proof will sometimes require prior declarations that some functions are respectful morphisms for some homogeneous relations. For our purpose, we need to generalize these rules to heterogeneous relations.

As before, we take a theorem and a goal as arguments and we must produce a proof of `thm → goal`, that is `impl thm goal`. We borrow the notation

$$\Gamma \vdash \tau \rightsquigarrow_p^R \tau'$$

which means that given an environment Γ in which τ and τ' are well-defined, p is a proof of $R(\tau, \tau')$.

Initially, given a theorem $\Gamma \vdash \tau$ and a goal $\Gamma \vdash \tau'$, we want to find a judgement of the form:

$$\Gamma \vdash \tau \rightsquigarrow_p^{\texttt{impl}} \tau'$$

[5] Here is the proof as a Coq development:

```
Theorem surj_decl :
  ∀ (A B : Type) (f : A → B) (g : B → A), (∀ x, f (g x) = x) →
  let R x x' := f x = x' in
  ((R ##> impl) ##> impl) (@all A) (@all B).
Proof.
  lazy delta zeta.
  intros A B f g H1 R R' H2 H3 x'.
  apply H2 with (x := g x'); trivial.
Qed.
```

Rules. We give in Fig. 1 the rules to get to that judgement, adapted from [6]. We have dropped the UNIFY rule as it was used for rewriting but does not apply in our case. To avoid unnecessary complexity, we have also chosen to drop the SUB rule in a first version.

$$\frac{p : R(\tau, \tau') \in \Gamma}{\Gamma \vdash \tau \rightsquigarrow^R_p \tau'}\ \textsc{Env} \qquad \frac{p : R(\tau, \tau') \in \text{Tables}}{\Gamma \vdash \tau \rightsquigarrow^R_p \tau'}\ \textsc{Table}$$

$$\frac{\Gamma, x : \tau_1, x' : \tau_1', H : R(x, x') \vdash \tau_2 \rightsquigarrow^S_p \tau_2'}{\Gamma \vdash \lambda x : \tau_1 . \tau_2 \rightsquigarrow^{R\ \#\#>\ S}_{\lambda x:\tau_1, x':\tau_1', H:R(x,x').p} \lambda x' : \tau_1' . \tau_2'}\ \textsc{Lambda}$$

$$\frac{\Gamma \vdash f \rightsquigarrow^{R\ \#\#>\ S}_{p_f} f' \quad \Gamma \vdash e \rightsquigarrow^R_{p_e} e'}{\Gamma \vdash f(e) \rightsquigarrow^S_{p_f(e,e',p_e)} f'(e')}\ \textsc{App}$$

$$\frac{\Gamma \vdash \texttt{@all}\ \tau_1\ (\lambda x : \tau_1 . \tau_2) \rightsquigarrow^R_p \texttt{@all}\ \tau_1'\ (\lambda x' : \tau_1' . \tau_2')}{\Gamma \vdash \forall x : \tau_1, \tau_2 \rightsquigarrow^R_p \forall x' : \tau_1', \tau_2'}\ \textsc{Forall}$$

$$\frac{\Gamma \vdash \texttt{impl}\ \tau_1\ \tau_2 \rightsquigarrow^R_p \texttt{impl}\ \tau_1'\ \tau_2'}{\Gamma \vdash \tau_1 \to \tau_2 \rightsquigarrow^R_p \tau_1' \to \tau_2'}\ \textsc{Arrow}$$

Fig. 1. `exact modulo` inference rules.

From these rules, we plan to derive a deterministic algorithm, which we will implement and test.

We will now illustrate each of these rules by a few examples, taken from the transfer of Axiom 6 (transitivity of $\leq_{\text{nat}}$) to Theorem 6 (transitivity of $\leq_{\text{N}}$).

Example 5. Initially, we want to find a judgement of the form

$$\begin{aligned} \vdash \quad & \forall x, y, z \in \text{nat},\ x \leq_{\text{nat}} y \ \Rightarrow\ y \leq_{\text{nat}} z \ \Rightarrow\ x \leq_{\text{nat}} z \\ \rightsquigarrow^{\texttt{impl}} \quad & \forall x', y', z' \in \text{N},\ x' \leq_{\text{N}} y' \ \Rightarrow\ y' \leq_{\text{N}} z' \ \Rightarrow\ x' \leq_{\text{N}} z' \ . \end{aligned}$$

By rule FORALL, this reduces to

$$\begin{aligned} \vdash \quad & \texttt{@all}\ \text{nat}\ (\lambda x : \text{nat},\ \forall y, z \in \text{nat},\ x \leq_{\text{nat}} y \Rightarrow\ y \leq_{\text{nat}} z \Rightarrow\ x \leq_{\text{nat}} z) \\ \rightsquigarrow^{\texttt{impl}} \quad & \texttt{@all}\ \ \text{N}\ (\lambda x' : \text{N},\ \ \forall y', z' \in \text{N},\ x' \leq_{\text{N}} y' \Rightarrow\ y' \leq_{\text{N}} z' \Rightarrow\ x' \leq_{\text{N}} z') \ . \end{aligned}$$

By rule APP, this reduces to

$$\begin{aligned} \vdash \quad & \lambda x : \text{nat},\ \forall y, z \in \text{nat},\ x \leq_{\text{nat}} y \Rightarrow\ y \leq_{\text{nat}} z \Rightarrow\ x \leq_{\text{nat}} z \\ \rightsquigarrow^{R} \quad & \lambda x' : \text{N},\ \ \forall y', z' \in \text{N},\ x' \leq_{\text{N}} y' \Rightarrow\ y' \leq_{\text{N}} z' \Rightarrow\ x' \leq_{\text{N}} z' \ , \end{aligned} \tag{8}$$

$$\vdash \quad \texttt{@all}\ \text{nat} \rightsquigarrow^{R\ \#\#>\ \texttt{impl}} \quad \texttt{@all}\ \text{N} \ . \tag{9}$$

Then (9) is solved by applying rule TABLE. We get $R =$ `natN ##> impl` . Finally, we can report the value of R in (8) and apply rule LAMBDA and thus our

initial problem reduces to

$$x : \mathrm{nat}, x' : \mathrm{N}, H : \texttt{natN}\ x\ x' \vdash \forall y, z \in \mathrm{nat},\ x \leq_{\mathrm{nat}} y\ \Rightarrow\ y \leq_{\mathrm{nat}} z\ \Rightarrow\ x \leq_{\mathrm{nat}} z$$
$$\rightsquigarrow^{\texttt{impl}} \forall y', z' \in \mathrm{N},\ x' \leq_{\mathrm{N}} y'\ \Rightarrow\ y' \leq_{\mathrm{N}} z'\ \Rightarrow\ x' \leq_{\mathrm{N}} z'\ .$$

From now on, $\Gamma = x : \mathrm{nat}, x' : \mathrm{N}, H : \texttt{natN}\ x\ x', y : \mathrm{nat}, y' : \mathrm{N}, H_1 : \texttt{natN}\ y\ y', z : \mathrm{nat}, z' : \mathrm{N}, H_2 : \texttt{natN}\ z\ z'$. We now consider the problem of finding a judgement of the form

$$\begin{aligned} \Gamma \vdash\ & x \leq_{\mathrm{nat}} y \Rightarrow\ y \leq_{\mathrm{nat}} z \Rightarrow\ x \leq_{\mathrm{nat}} z \\ \rightsquigarrow^{\texttt{impl}}\ & x' \leq_{\mathrm{N}} y' \Rightarrow\ y' \leq_{\mathrm{N}} z' \Rightarrow\ x' \leq_{\mathrm{N}} z'\ . \end{aligned}$$

By rule Impl, this reduces to

$$\begin{aligned} \Gamma \vdash\ & \texttt{impl}\ (x \leq_{\mathrm{nat}} y)\ (y \leq_{\mathrm{nat}} z \Rightarrow\ x \leq_{\mathrm{nat}} z) \\ \rightsquigarrow^{\texttt{impl}}\ & \texttt{impl}\ (x' \leq_{\mathrm{N}} y')\ (y' \leq_{\mathrm{N}} z' \Rightarrow\ x' \leq_{\mathrm{N}} z')\ . \end{aligned}$$

By rule App, this reduces to

$$\Gamma \vdash y \leq_{\mathrm{nat}} z \Rightarrow x \leq_{\mathrm{nat}} z \rightsquigarrow^{R} y' \leq_{\mathrm{N}} z' \Rightarrow x' \leq_{\mathrm{N}} z'\ , \tag{10}$$

$$\Gamma \vdash \texttt{impl}\ (x \leq_{\mathrm{nat}} y) \rightsquigarrow^{R\#\#>\texttt{impl}} \texttt{impl}\ (x' \leq_{\mathrm{N}} y')\ . \tag{11}$$

By rule App, (11) reduces again to

$$\Gamma \vdash x \leq_{\mathrm{nat}} y \rightsquigarrow^{S} x' \leq_{\mathrm{N}} y'\ , \tag{12}$$

$$\Gamma \vdash \texttt{impl} \rightsquigarrow^{S\#\#>R\#\#>\texttt{impl}} \texttt{impl}\ . \tag{13}$$

We will make sure that the tables are pre-filled so that judgements such as (13) can be solved with rule Table. In that case, we will get $S = \texttt{impl}^{-1}$ and $R = \texttt{impl}$. Now by rule App, (12) reduces to

$$\Gamma \vdash y \rightsquigarrow^{T} y'\ , \tag{14}$$

$$\Gamma \vdash \texttt{le}\ y \rightsquigarrow^{T\#\#>\texttt{impl}^{-1}} \texttt{N.le}\ y'\ . \tag{15}$$

Rule Env allows us to derive (14) with $T = \texttt{natN}$.

As for (15), it can be solved after a few more steps by using the knowledge that ($\texttt{natN ##> natN ##> impl}^{-1}$) `le N.le` which will be one of the user-provided transfer lemmas (in that case, it corresponds to Axiom 4). Therefore, there only remains to solve (10) in ways similar to this example.

4 Related work

4.1 Proof reuse

More than ten years ago, Nicolas Magaud [3] proposed an extension of Coq that seemed to share our objectives. Notably, he was able to transfer all the theorems that were, at the time, in the standard Arith library, from `nat` to `N`.

The approach was quite intricate because it was able to transfer proofs, and not just theorems. Given two isomorphic data-types, one will be considered as the *origin type* and the other one as the *target type*. The first step is to define functions to model the origin constructors within the target type. Moreover, new recursion operators behaving like the ones of the origin type are added to the target type.

With such a projection of the origin type into the target type, it is easy to project operators and relations. Proofs are transferred in the same way. The last step is to establish extensional equality between projected operators and the corresponding native operators of the target type.

While interesting, we do not need to take such a complicated path for our objective which is only *theorem reuse*. Using Magaud's approach requires much more work in establishing the relations between the two data-types. Moreover, our approach is more powerful in a sense: we can transfer properties between two data-types even if we know nothing of their content and the transfer lemmas where provided as axioms.

4.2 Algorithm reuse

A much more recent work by Cohen et al [2] has been of much inspiration to us. However, the focus is not the same. The authors are thinking in term of program verification and algorithm reuse through parametricity when refining proof-oriented data-types into efficient computation-oriented data-types. Parametricity then enables the automatic transfer of algorithm correctness proofs.

A main difference showing that their work and our work are complementary rather than overlapping lies in that they typically allow refined types to contain more objects, including objects which would have no meaning (no specification). We require precisely the opposite so as to be able to translate theorems stating properties *for all* elements, including unicity properties. However, as our second algorithm will accept other equivalent relations than just Leibniz equality, we will still be able to work with types in which the same object has many equivalent constructions.

5 Conclusion

In this paper, we have shown how a simple algorithm can make use of a few initial declarations to ease the reuse of results from one data-type to another.

As we improve our algorithm and become able to transfer more theorems, we will still have a lot to do in order to make our plugin as simple-to-use as possible.

A first easy step will be to transform our `exact modulo` tactic into an `apply modulo` tactic.

Then, we will need to allow for compositionality in ways similar to [2]. First, by allowing and handling transfer declarations for parametrized types. Then, by finding paths from one type to another, even when the relation between the

two was not declared, but can be established by going through a sequence of transfers.

We view this work as a little but quite interesting step in the enormous task of making the use of a formal proof system as easy as a pen-and-paper proof.

References

1. Jacek Chrzaszcz. Implementing modules in the coq system. In *Theorem Proving in Higher Order Logics*, pages 270–286. Springer, 2003.
2. Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free! In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8307 LNCS, pages 147–162, 2013.
3. Nicolas Magaud. Changing data representation within the coq system. In *Theorem Proving in Higher Order Logics*, pages 87–102. Springer, 2003.
4. Coq development team. *The Coq proof assistant reference manual*. Inria, 2015. Version 8.5.
5. Christine Paulin-Mohring. *Inductive definitions in the system coq rules and properties*. Springer, 1993.
6. Matthieu Sozeau. A new look at generalized rewriting in type theory. *Journal of Formalized Reasoning*, 2(1):41–62, 2010.