Internship report: transfer of theorems along isomorphisms in the Coq proof assistant

Théo Zimmermann, supervised by Hugo Herbelin,
PPS laboratory, Paris-Diderot University

March–July 2015

General Context

Machine-checked proofs, that is proofs which are stated in such a language that a computer can check their validity, also known as formal mathematics, have been a field of study for over forty years [3] but have only recently started to reach a large community of users. Interactive proof assistants, which are software to help write such proofs, are of interest both to the verification community (software verification, hardware verification) and to the mathematical community. The latter has recently produced proofs of such size that it is now fearing that the peer-review process alone is not enough to guarantee the correctness of a proof [5, 15]. Despite having reached a certain level of maturity, proof assistants are still hard to use and there is on-going effort to lower the barrier of entry and allow a much larger community of users to grow.

Problem Studied

In mathematics, it is common practice to have several equivalent definitions for the same object. Mathematicians will identify them up to isomorphism and will not worry later on which construction they use, as theorems proved for one construction will be valid for all. Correspondingly, in the formal mathematics area, it is common to see several data-types representing the same objects. My internship aimed at making the use of several isomorphic constructions as simple and as transparent as it can be done informally in mathematics. This requires automatically inferring and inserting the missing proof-steps.

Proposed Contributions

I worked from examples to understand the required capabilities. I incrementally developed a theory of theorem transfer between related types and I devised algorithms which could actually carry the transfer out. I implemented these algorithms as two successive prototypes, the first one as a self-contained Coq plugin, the second one as a Coq library relying heavily on Coq's type-class mechanism [14]. Both are available on the web at `https://github.com/Zimmi48/transfer` and are freely reusable. I tested both implementations on various examples.

Arguments Supporting Their Validity

The first algorithm and prototype could already be used to transfer a restricted but interesting set of theorems, not only along isomorphisms but also, in some cases, along surjective morphisms. The second algorithm and prototype lifts a large number of limitations by allowing the transfer of all sorts of theorems and also accepts to

work with much weaker relations between types. I have used it to transfer many theorems of Coq's standard library between the two existing representations for natural numbers, unary numbers `nat` (well-suited for reasoning) and binary numbers `N` (better suited for computation). I have also explored less obvious applications such as changes of representation from natural numbers to sets in order to do combinatorial proofs of arithmetic identities (inspired by [9]).

The theoretical framework is supported by the existence of many similar ideas by a lot of researchers working on close subjects. I had been inspired by some of their articles [2, 13] but I also discovered some connections later on, most noticeably with [7], whose theory looks very similar to the one I developed. There is also existing mathematical research on close subjects such as relational algebra [12] and finally this theory also has deep connections to the notion of parametricity [10] which I have not had time to explore yet.

Summary and Future Work

Two ideas were sketched during this work and would probably be interesting to explore further. The first is to integrate this transfer theory more tightly with Coq's unification algorithm so as to propose a unification-modulo capability. The second is to explore the connections between theorem transfer and generalized rewriting [13]. These connections certainly exist as I have built upon ideas from generalized rewriting to come with my theory and final algorithm but I did not have enough time to explore them fully and a broader understanding could potentially yield even larger applications.

I will look very closely at all potential use of my work so that I can adapt it if new requirements arise or if problematic corner cases are discovered.

Other than that, I plan to focus on making proof-assistants easier to use for mathematicians in many other ways.

Notes and Acknowledgements

I have presented part of this work at the Conference on Intelligent Computer Mathematics 2015, in Washington D.C., and this report is partially built upon the article, cosigned with Hugo Herbelin, that was accepted for the work-in-progress proceedings of this conference. Consequently, the full report is also in English. Another consequence is that I consistently say "we" instead of "I" in what follows. But this is I who wrote alone the article and obviously the present report.

I would like to thank Hugo Herbelin who has been a great supervisor, by his constant attention, the long hours he took to discuss with me, helping with my research, giving me ideas, and even looking at my code and debugging with me (this last bit is remarkable enough to be worth noting).

I would also like to thank the anonymous reviewers of CICM 2015 for their helpful comments, as well as Matthieu Sozeau, Arnaud Spiwack and many others for their help, advice and the very interesting discussions I had with them. It was a pleasure meeting with a lot of motivated Coq developers and users at the Coq coding sprint and Coq workshop 2015 in Sophia-Antipolis then with a more diverse but as exciting community during CICM 2015.

1 Introduction

With examples such as the well-known relation between linear maps and matrices, or the various constructions of real numbers (equivalence classes of Cauchy sequences, Dedekind cuts, infinite sequences of digits, subset of complex numbers), we see that there are 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 construction but theorems apply to all.

In formal systems like Coq [4], a canonical example is the two constructions available for natural numbers. The most natural construction and the closer to the mathematical view is unary (`0`, `(S 0)` (successor of zero), `(S (S 0))` and so on) while the more efficient binary construction is closer 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 using the module and functor system [1] or using the type-class system [14]. While this has the advantage of factoring proofs, it can also make the proof harder as it does not allow taking advantage of the specifics of the implementation.

The purpose of this work is to make it easy to transport theorems to all isomorphic constructions even when the proof relies on one particular such construction. In an informal setting, the mathematician would declare that "we can identify the two structures" once she has proved they were isomorphic and would proceed from there. Our goal is to validate such claim by bringing in any justification that would be considered as missing by Coq's proof checker.

We can also see that our work will be useful for easily merging several independent library developments (for instance, there are already a lot of libraries for real numbers but they are based on various constructions and do not contain exactly the same theorems).

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 a weaker correspondence is sufficient. That is why we require elementary transfer declarations from the user rather than a single big declaration that two given structures are isomorphic.

Our algorithm will typically allow the following transfer:

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

Axiom 1 (A is empty)**.**

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

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

Theorem 1 (A' is empty)**.**

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

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

The next section will discuss in more details our objective as we will try to decide what sort of transfer we want to handle. Next, we will present each of our two prototypes (the ideas behind it, how to use it, how we implemented it and how we tested it). Finally, we will discuss related work and other work directions we only sketched.

2 What Is a Transfer?

To say that two structures are isomorphic, we first need to know what a structure is. A structure is generally accepted to be made of a base-set (or base-type[1]), which we often call *carrier*, and a number of operations and relations on the carrier. Additionally, we often associate some axioms to a given structure (for instance, a set with a single binary operation is called a *magma*, but if the operation is associative it is then called a *semi-group*).

2.1 How do we relate two structures?

Two structures are isomorphic if there exists a one-to-one correspondence (a bijection) between the two carriers which also relates their operations and relations (i.e. it is a morphism for these operations and relations).

Expressing this notion rigorously is not so easy but we find that decomposing it makes things simpler. Rather than asking for two structures and a big theorem showing that they are isomorphic, we will ask for a collection of transfer declarations, thus allowing to also express weaker correspondences: for instance two structures may not be completely isomorphic but there may be a one-to-one correspondence preserving a subset of their operations and relations; or there may be a correspondence preserving some operations but it may not be one-to-one.

That is why rather than asking for bijective functions between types, our first prototype asks for transfer functions with a right-inverse (i.e. retractions[2]). Given that our first prototype only considers relations (no operations or general functions), it will also require what we call transfer lemmas for these relations.

Our second prototype will go even further. It will not need transfer functions anymore but heterogeneous relations between the two data-types instead. As a reminder, a function can always be seen as a relation and for a relation to be a

[1] We work in a type-theory context, thus we will see sets and types as synonymous and we will more often use the latter designation.

[2] If we define surjective functions as functions whose image is the entirety of its codomain, being surjective is a weaker requirement than being a retraction – in the absence of the axiom of choice – and this should normally be sufficient but the existence of a right-inverse will be useful to have a simpler algorithm.

bijective function, it must have four independent properties: being right-unique (or functional), left-unique (or injective), left-total and right-total (or surjective). Using relations instead of functions provides a more modular way of relating two data-types and we will see that it has other advantages too.

3 First Prototype

For this first prototype, we are limiting ourselves to transferring first-order formulas containing only universal quantifiers, implication and relations.

3.1 User-provided declarations

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[3] 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 defining 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)**.**

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

[3] In other words, using terminology of category theory, we ask that g be a section of f and f be a retraction of g.

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

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

Finally, we are given the following result to transfer:

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

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

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 following two hypotheses 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 quantifier to transfer inside one of the hypotheses. This suggests a similar example where Axiom 2 would not hold, thus where there would be no isomorphism between the two related data-types. Such an example is provided in the repository containing the plugin: we transfer various theorems (such as transitivity of $\leq$) from $\mathbb{Z}$ to $\mathbb{N}$.

3.2 Preliminaries in type-theory-based logic

Understanding the proposed algorithm will not require much knowledge about the internals of Coq:

- Dependent products are the way in which the Calculus of Inductive Constructions [4, Ch. 4], the logical base of Coq, models both universal quantification and implication. The implication is just the degenerate non-dependent case, i.e. $A \Rightarrow B$ is just an abbreviation for $\forall x : A, B$ when x does not appear in B.
- In the Calculus of Inductive 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.

3.3 The algorithm

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 is recursive 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.

You will have noticed, at line 25 of Algorithm 1, 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 $\leq_{\mathrm{N}}$) while remembering that the right-hand side of an implication is covariant while the left-hand side is contravariant.

We could add 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 Inductive 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.

3.4 Implementation and tests

This algorithm was implemented as a Coq plugin, written in OCaml.

Some choices which were previously described were actually made to ease the implementation work. In particular, the requirement that there is only one transfer function between two types allows us to avoid any form of backtracking. The need for a right-inverse for all transfer functions and the fact that we predict exactly where to substitute x' with $f(g(x'))$ means that we do not need to remember which surjection lemmas were used.

Our working cases were theorems such as the transitivity of a relation. We have also tested the transfer of a few theorems from $\mathbb{Z}$ to $\mathbb{N}$. However, most theorems in Coq's standard library cannot be transferred with this plugin as they very often contain operations, functions or equality.

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 $F = R(t_1, \ldots, t_n)$ **and** $F' = R'(t'_1, \ldots, t'_n)$ **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 $t'_i \neq f(t_i)$ **then**
        return failure
    return $\rho_{\text{transfer}}(t_1, \ldots, t_n, \rho_F)$
  else if $F = \forall x : A, B$ **and** $F' = \forall x' : A', B'$ **then**
    $\Gamma \leftarrow \Gamma, x' : A'$
    $t \leftarrow$ ExactModulo(Γ, A', A, x')
    if $t \neq$ failure **then**
      $\rho_{\text{rec}} \leftarrow$ ExactModulo$(\Gamma, B, B', \rho_F(t))$
        ▷ **return** failure if $\rho_{\text{rec}} =$ failure
      return $\lambda x' : A'.\, \rho_{\text{rec}}$
    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 $\lambda x' : A'.\, \rho_{\text{rec}}$ 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

3.4.1 Using the plugin.

Declarations have the following form:

- `Declare Surjection f by (g, proof).`

  where `f`, `g` and `proof` are previously declared identifiers such that `f : A` $\rightarrow$ `B`, `g : B` $\rightarrow$ `A` and `proof` : $\forall$ `x : B, f (g x) = x`.

- `Declare Transfer r to r' by (f, proof).`

  where `r`, `r'`, `f` and `proof` are previously declared identifiers such that `r` has n arguments of type `A` and `r'` has n arguments of type `B`, `f : A` $\rightarrow$ `B` and `proof` : $\forall$ `x1 ... xn : A, r x1 ... xn` $\rightarrow$ `r' (f x1) ... (f xn)`.

We provide a new tactic called `exact modulo`. It can be used within any proof development, very similarly to how the `exact` tactic would be used (the `exact` tactic is used to give directly a proof-term for the current goal, for instance the name of a theorem whose statement matches exactly the goal).

4 Second Prototype

We list here the most problematic limitations of the first prototype. Fortunately, the second prototype will be able to lift them all.

Functions. So far we have considered only relations. Even though any function can be expressed as a relation, this path would require a lot of preliminary rewriting steps; thus 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, as well as internal operators, would be supported.

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. In the first prototype, 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 [13] to allow rewriting with other equivalence relations and this has been our main source of inspiration for the second prototype.

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 not require such a strong declaration anymore.

Multiple correspondences. There might exist several isomorphism between two types but our first prototype prevented to declare them all at once. There might even be more non-isomorphic correspondences that are worth stating and working with. Because our second prototype will be based upon the exploration of a search space (with backtracking, instead of a linear decomposition algorithm), this limitation will be lifted as well.

4.1 Generalized declarations

As we announced before, we are now replacing transfer functions with relations. This will allow finer-grained declarations and the unification of what we previously qualified as transfer lemmas or surjection lemmas.

4.1.1 Transfer lemmas.

Coq's `Morphisms` library[4] introduces a 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.

We will note

```
(R ##> R') f g := ∀ (x : X) (y : Y), R x y → R' (f x) (g y) .
```

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

$$\begin{array}{ccc} X & \xleftrightarrow{R} & Y \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle g} \\ X' & \xleftrightarrow{R'} & Y' \end{array}$$

It is easy to show that this corresponds precisely to a very general notion of homomorphism that can be found in mathematical textbooks such as [12, Ch. 5.7]. The pair of mappings (f, g) is a homomorphism between the two "structures" $(X \times Y, R)$ and $(X' \times Y', R')$ if the following holds:

$$R \circ g \subseteq f \circ R'$$

where $\circ$ is the relational composition, i.e.

$$\forall x \in X, y' \in Y', [(R \circ g)(x, y') \Leftrightarrow \exists y \in Y, R(x, y) \wedge g(y) = y'] \quad ,$$

$$\forall x \in X, y' \in Y', [(f \circ R')(x, y') \Leftrightarrow \exists x' \in X', f(x) = x' \wedge R'(x', y')] \quad .$$

It will be possible to declare all sorts of transfer lemmas thanks to the respectful arrow as can be seen in the following example.

Example 3. *Let us consider a 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'.
```

[4] Coq's `Morphisms` library is part of Sozeau's work [13] 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`.

Then, we can declare how to transfer various functions and relations[5]*:*

```
Instance le_transfer : Related (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 the syntax* `##>` *associates to the right by default). That is, after unfolding the definitions of* `Related`*,* `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:*

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

where `@eq bool` *is the Boolean equality.*

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

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

4.1.2 Inherent properties of the relation (ex-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 .
```

Any surjection declaration in the style of Sec. 3:

```
Declare Surjection f by (g, proof).
```

can be equivalently replaced by the following three declarations:

```
Instance R_surj :
    Related ((R ##> impl) ##> impl) (@all A) (@all A').
Instance R_tot :
    Related ((R ##> impl^-1) ##> impl^-1) (@all A) (@all A').
Instance R_func : Related (R ##> R ##> impl) (@eq A) (@eq A').
```

where `R x x' := f x = x'` and `impl^-1 A B := B → A` .

The first declaration corresponds to the surjectivity of relation R (also called right-totality). The second and third declaration express the fact that R is a mapping. More precisely, the second declaration expresses the (left-)totality of R while the third declaration expresses the knowledge that R is functional (also called univalent in [12, Ch. 5.1] or right-unique elsewhere).

The three declarations provide interesting "point-free" formulations of a relation totality and unicity properties. Let us unfold two of them to give more intuition on what they mean:

[5] These declarations are written as instances of a type-class called `Related`: `Related R x y := R x y`. This comes from the way the second prototype has been implemented and the use of type-class instance declarations instead of ad-hoc declarations as with the first prototype.

```
Theorem R_surj :
    ∀ P P', (∀ (x : A) (x' : A'), R x x' → P x → P' x') →
    (∀ x : A, P x) → ∀ x' : A', P' x'.
Theorem R_func :
    ∀ (x : A) (x' : A'), R x x' →
    ∀ (y : A) (y' : A'), R y y' → x = y → x' = y'.
```

We immediately see that `R_func` indeed expresses that R is functional (each input has at most one output). As for `R_surj`, while it is clearly a necessary condition for surjectivity, we will have to instantiate the theorem with `P = λ _ : A, True` and `P' = λ x' : A', ∃ x : A, R x x'` to see that it is sufficient.

This new and more precise way of replacing surjection declarations brings at least two advantages. First, as it is more general, it will allow considering data-types which are related by a non-functional or non-total relation. Second, by replacing `eq` by any partial equivalence relation and `all` by any bounded quantification, we make possible to relate two partial quotients and not only two classic data-types.

4.2 Transfer to the context

In [13], 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 derive a judgment of the form:

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

4.2.1 Rules.

We give in Fig. 1 the rules to get to that judgment, adapted from [13]. 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[6].

The Lambda rule from [13] used a special pointwise equivalence relation between functions. We have found that its heterogeneous generalization corresponds exactly to our heterogeneous respectful arrow. This gives rules Lambda and App an air of introduction and elimination rules as in typing inference algorithms.

[6] A lot of complexity of the generalized rewriting algorithm comes from the handling of subrelations. This capability brings several advantages such as reducing the number of required declarations. However, it needs to be dealt with a lot of care in order to avoid a combinatorial explosion of the search space. We wanted to produce a running prototype quickly, hence we could not deal with that kind of increased complexity. We describe in subsection 4.3 the trick we used to transfer theorems containing a if-and-only-if anyway.

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

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

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

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

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

Figure 1: Inference rules for the transfer to the context.

Before explaining how we implemented our second prototype based on these rules, we illustrate each of them 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 4. *Initially, we want to find a judgment 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*

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

By rule App*, this reduces to*

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

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

Then (9) is solved by applying rule Table*. We get* $R = \texttt{natN ##> impl}$ *. Finally, we can report the value of* R *in (8) and apply rule* Lambda *and thus our initial problem reduces to*

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

From now on,

$$\begin{aligned} \Gamma \ = \ & x : \text{nat},\ x' : \text{N},\ H : \texttt{natN}\ x\ x', \\ & y : \text{nat},\ y' : \text{N},\ H_1 : \texttt{natN}\ y\ y', \\ & z : \text{nat},\ z' : \text{N},\ H_2 : \texttt{natN}\ z\ z' \quad . \end{aligned}$$

We now consider the problem of finding a judgment of the form

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

By rule Impl*, this reduces to*

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

By rule App*, this reduces to*

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

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

By rule App*, (11) reduces again to*

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

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

We will make sure that the tables are pre-filled so that judgments 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*

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

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

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 `(natN ##> natN ##> impl`$^{-1}$`) le N.le`*, which will be one of the user-provided transfer lemmas (it corresponds to Axiom 4). Therefore, there only remains to solve (10) in ways similar to this example.*

4.3 Implementation

In line with Sozeau's work, we are using type-classes instance resolution to solve the search problem that these inference rules create.

We first define a specific type-class `Related R x y := R x y` which will be useful to store both the various transfer declarations (as was shown with the examples in subsection 4.1) but also the generic declarations[7] and the inference rules[8].

To launch instance resolution, we use the very general theorem `modulo` whose type is the following:

```
modulo : ?t → ?u
where
?t : [ |- Prop]
?u : [ |- Prop]
?class : [ |- Related impl ?t ?u]
```

[7] Such as `Instance impl1 : Related (impl`$^{-1}$ `##> impl ##> impl) impl impl.`

[8] To the rules that were presented earlier, we added a last one to treat if-and-only-if specifically. This rule basically unfolds its definition to a conjunction of implications.

That is: for any two propositions `?t` and `?u`, we can show that the former entails the later by finding an instance of `Related impl ?t ?u`. This is what was noted earlier as

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

with τ for `?t`, τ' for `?u` and p for `?class`.

Since we rely on the type-class instance search, we basically just have to define the right instances (in particular to represent the inference rules). This is what we do and the first part of our implementation only takes 80 lines of Coq's vernacular. The full library is less than 250 lines and it additionally contains generic and useful declarations (for instance, it contains a declaration for the $\wedge$ connective and another one for the $\vee$ connective) and the two following theorems showing the link between the way we ask for left-total/right-total declarations and what it means in terms of $\forall$ and $\exists$:

```
Theorem surj_decl :
    ∀ (A B : Type) (R : A → B → Prop),
    (∀ x' : B, ∃ x : A, R x x') ↔
    ((R ##> impl) ##> impl) (@all A) (@all B).
Theorem tot_decl :
    ∀ (A B : Type) (R : A → B → Prop),
    (∀ x : A, ∃ x' : B, R x x') ↔
    ((R ##> impl⁻¹) ##> impl⁻¹) (@all A) (@all B).
```

The two theorems can be used in proofs of subsequent declarations as can be seen in the following example:

```
Instance surjectivity :
      Related ((R ##> impl) ##> impl) (@all _) (@all _).
Proof.
    split. (* ((R ##> impl) ##> impl) (@all _) (@all _) *)
    apply surj_decl.               (* ∀ x', ∃ x, R x x' *)
    ...
```

Being very general, `modulo` will normally be used only in one of the two following ways, thus providing enough information to bound the space search to something reasonable.

4.3.1 Transfer of theorem.

This is the standard use. It corresponds to the `exact modulo` tactic of the first prototype. Given a theorem `thm` and a proof goal corresponding structurally to this theorem, but with some type transformations, and supposing that the required declarations were made, the user can call `exact (modulo thm)` to solve the current goal. This means that both `?t`, the theorem, and `?u`, the goal, will be known. Thus they will guide instance resolution.

If the tactic fails, it is likely due to missing declarations.

4.3.2 Change of representation.

This other use occurred to us after the implementation had been finished already and it was inspired by Isabelle's `transfer'` tactic [7]. This tactic uses the transfer

declarations to change the current goal to a new goal where the considered data-types have been replaced by others and the operators and relations likewise. Raggi et al [9] show how informal mathematical proofs often do such a change of representation, either explicitly or implicitly. Thus, having such a tactic can help deriving a formal proof that looks similar to the informal one.

Without changing our implementation, we remarked that such a use was possible by calling `apply modulo` in a proof context where none of the variables or hypotheses to transfer have yet been introduced (because `apply modulo` can change the current goal but not the context).

In that case, only `?u`, the current goal, is known. The new goal `?t` will be inferred from the transfer declarations, together with a proof of `Related impl ?t ?u`. As it creates a more complicated search problem, this use can fail for many reasons:

It can leave the goal unchanged. If this happens, it means that there are too many transfer declarations. Technically, it is true that `Related impl ?u ?u` but this is obviously not what we want. We have restricted our library so that it does not include facts that would systematically prevent `apply modulo` to be used in an interesting way but this also means that `exact (modulo thm)` can fail if some things need to be left unchanged. Here is an example of a declaration we have removed from the library in that aim:

```
Instance impl2 : ∀ (A : Prop), Related impl A A.
```

There clearly is a trade-off to be have between the power of `exact (modulo thm)` and the power of `apply modulo`. A solution could be to have separate libraries of instance declarations for these two uses.

It can make the wrong change of representation. This unwanted behavior would also come from having too many declarations. The change of representation would still be logically sound but would not be expected and would not necessarily be helpful. Indeed, there is no way to say which transfer we are interested in, if we have previously declared several. To our knowledge, Isabelle's `transfer'` tactic does not provide any way to guide the change either. It would be an interesting further work to find a way of providing the domain targeted by the change of representation, as it could not only solve this problem but would also avoid leaving the goal unchanged.

It always fails on theorem of the form `∀ x, a = b.` This was an unexpected error which is a lot more frustrating given that there are many theorems of that form. It may be understood by looking carefully at the trace of the instance resolution but while it would be too long to explain here, we can merely remark that the problem finds its source in a too loose integration between the instance resolution algorithm and the unification algorithm. Both algorithms can branch when there are several options and backtrack if they have taken a dead-end. However, a dead-end in the instance resolution mechanism will not make the unification algorithm backtrack even if it still has other ways of unifying its arguments.

There is a trick to make `apply modulo` work however: it is to add a dummy hypothesis such as `True`, thus changing the goal to `∀ x, True → a = b`.

4.4 Tests

As an evaluation, we decided to see how many theorems we would be able to transfer from `nat` to `N`.[9]

Coq's `NArith.Nnat` library contains a lot of theorems relating `nat` and `N`. We devised a tactic to automatically transform these theorems in transfer declarations that our library can use. Additionally, we added a few essential declarations that had no corresponding theorem in this library (in particular, the surjectivity of the transfer function).

With these declarations, we were able to automatically transfer most theorems in Coq's `PeanoNat` library. The only reason we could not transfer some theorems was a lack of transfer declarations for the relations or functions they contained.

Mathematically meaningful changes of representation. At CICM 2015, Daniel Raggi presented his work [9] on the use of Isabelle's Transfer package to produce proofs that look closer to the ones a mathematician would write. Two of his working cases were the representation of natural numbers as bag of primes (prime factor representation of a natural number) and their representation as sets of some cardinal (in order to do combinatorial proofs).

We have tried to translate his `SetNat.thy` file to Coq, using our library. A first problem was to choose among the various libraries for finite sets that have been developed in Coq. Most of these libraries are using the module/functor system to be generic but that makes them quite hard to use. Besides, none of the libraries we found contained functions such as the *powerset* and its restriction to subsets of a specific cardinal. This already ruled out the most interesting combinatorial proofs about the *choose* function.

The last difficulty we encountered was not specific to Coq or our transfer library as it was already there in the Isabelle file we translated: a sum on natural numbers corresponds to a disjunct union of sets. However, no such function was already defined and while it can be, that means adding another argument which is a proof of disjunctness. In that case, the two functions (addition and disjunct union) do not have the same number of arguments anymore and it becomes very difficult to relate them. The proposed solution was to represent these two functions as relations and while this works indeed, this is a little frustrating because it makes the use of the transfer capability more involved (this was precisely the kind of things we thrive to avoid with that second prototype).

5 Related Work

5.1 Proof reuse

More than ten years ago, Nicolas Magaud [8] 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

[9] All the tests are available with the library in the repository.

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.

5.2 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. In the context of program verification, the authors propose a general method for 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. Although they give this general method, they explain why they do not provide a generic implementation. Our focus being on transparency and usability by mathematicians, we decided to create such an implementation.

An other inspiring characteristic of their work lies in that they typically allow refined types to contain more objects, including objects which would have no meaning (no specification). We generally require precisely the opposite so as to be able to translate theorems stating properties *for all* elements, including unicity properties, but as we remarked earlier our implementation would support declarations replacing `all` with bounded quantification. Bounded quantification would be useful for transferring theorems from a subset type to the corresponding elements of a larger type (for instance from $\mathbb{N}$ to the non-negative elements of $\mathbb{Z}$).

5.3 Other works proposing a heterogeneous respectful arrow

While Cohen et al. [2] inspired us to use a generalized heterogeneous respectful arrow to allow for more precise transfer declarations and remove the limitations of Algorithm 1, there are many other (and sometimes older) works proposing the same definition. One example of such a work is [6, Def. 13]. But this is not surprising as we have remarked in Sec. 4.1 that this arrow just encodes for an already existing mathematical notion of homomorphism.

Huffman and Kunčar [7] go further as they also show how the relational unicity and totality properties can be expressed in terms of the respectful arrow. They produced a Transfer package for Isabelle/HOL with comparable objectives to ours, and their `transfer` tactic is based on a two-step algorithm sharing many ideas with Sozeau's algorithm [13]. Nothing going as far as their Transfer package had yet been created for Coq however.

6 Discussion and Conclusion

Limitations. While we have successfully produced a very useful way of working with several representations, there is still a huge room for improvement. The current version of the type-class based library has very bad error-reporting. It does not handle subrelations, although that would be useful to reduce the number of required declarations. There is also a problem if we want to do transfers between representations in both directions: again, it requires a lot of new transfer declarations, some of them being just restatements of already valid ones[10]. Likewise, we do not handle yet composition of transfer declarations.

Unification modulo. Another work direction which we had started to explore at the beginning of this internship would have allowed for much more powerful tactics. This direction was to modify the unification algorithm (which is useful during the full elaboration of terms) to succeed not only when two terms are actually convertible but also when they are equivalent (or in a relation). In that case, the algorithm would also produce a proof of the equivalence and this proof would reduce to reflexivity when the equivalence relation is equality. This work direction would be similar to the type-class [14] and coercion mechanisms [11] in that it would make writing proofs easier and would leave a lot of disambiguation work to the elaboration mechanism. Although we have abandoned this idea for the time of the internship, in particular because diving in the already very complex unification algorithm of Coq was very hard and time-consuming, we still feel like it is a direction worth exploring.

Building up from generalized rewriting. Once we had the basic ideas for our second prototype, we felt like we could tweak the rewriting mechanism to handle heterogeneous relations as well and work on our implementation from there. It appeared again to be too complicated for the available time-frame but it is very obvious that there are a lot of similarities between the rewriting mechanism of Coq and our transfer capabilities. The similarities are even greater in the "change of representation" mode which can be considered as a sort of rewriting of the proof goal. We will probably benefit from exploring these similarities, both on a theoretical level and on an implementation level. It may even reveal novel and unforeseen applications.

Backtracking against linearity. While more powerful, our second prototype is also more complex than the first. The exploration of a search space may take an exponential time and we are not safely guarded against this as we allow the user to add new instances. In practice however, we have always observed reasonable speed (less than a second), unless a mistake was made which sent the search in an infinite loop. This kind of problem is very common when working with type-classes.

[10] For instance, from
`Instance inj_add : Related (natN ##> natN ##> natN) Nat.add N.add.`
we can deduce
`Instance inj_add' : Related (natN$^{-1}$ ##> natN$^{-1}$ ##> natN$^{-1}$) N.add Nat.add.`
but the second declaration is still required if it is to be used for a transfer from `N` to `nat`.

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 Dénes, and Anders Mörtberg. Refinements for free! In *Certified Programs and Proofs*, pages 147–162. Springer, 2013.

[3] Nicolaas Govert de Bruijn. The mathematical language Automath, its usage, and some of its extensions. In *Symposium on automatic demonstration*, pages 29–61. Springer, 1970.

[4] Coq development team. *The Coq proof assistant reference manual*. Inria, 2015. Version 8.5 beta.

[5] Georges Gonthier. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[6] Peter V Homeier. A design structure for higher order quotients. In *Theorem Proving in Higher Order Logics*, pages 130–146. Springer, 2005.

[7] Brian Huffman and Ondřej Kunčar. Lifting and transfer: A modular design for quotients in Isabelle/HOL. In *Certified Programs and Proofs*, pages 131–146. Springer, 2013.

[8] Nicolas Magaud. Changing data representation within the Coq system. In *Theorem Proving in Higher Order Logics*, pages 87–102. Springer, 2003.

[9] Daniel Raggi, Alan Bundy, Gudmund Grov, and Alison Pease. Automating change of representation for proofs in discrete mathematics. In *Conference on Intelligent Computer Mathematics*, volume 9150, pages 227–242. Springer, 2015.

[10] John C Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513—-523, 1983.

[11] Amokrane Saïbi. Typing algorithm in type theory with inheritance. In *Principles of Programming Languages*, pages 292–301, 1997.

[12] Gunther Schmidt. *Relational mathematics*, volume 132 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 2011.

[13] Matthieu Sozeau. A new look at generalized rewriting in type theory. *J. Formalized Reasoning*, 2(1):41–62, 2009.

[14] Matthieu Sozeau and Nicolas Oury. First-class type classes. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5170 LNCS:278–293, 2008.

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