Reflexive Tactics for Algebra, Revisited

Kazuhiko Sakaguchi ✉ 🆔
Unaffiliated researcher, Tokyo, Japan

─── **Abstract** ───

Computational reflection allows us to turn verified decision procedures into efficient automated reasoning tools in proof assistants. The typical applications of such methodology include decidable algebraic theories such as equational theories of commutative rings and lattices. However, such existing tools are known not to cooperate with *packed classes*, a methodology to define mathematical structures in dependent type theory, that allows for the sharing of vocabulary across the inheritance hierarchy. Moreover, such tools do not support homomorphisms whose domain and codomain types may differ. This paper demonstrates how to implement reflexive tactics that support packed classes and homomorphisms. As applications of our methodology, we adapt the `ring` and `field` tactics of Coq to the commutative ring and field structures of the Mathematical Components library, and apply the resulting tactics to the formal proof of the irrationality of $\zeta(3)$ by Chyzak, Mahboubi, and Sibut-Pinote. As a result, the lines of code in the proof scripts have been reduced by 8%, and the time required for proof checking has been decreased by 27%.

1 Introduction

Computational reflection [2] makes it possible to replace proof steps with computations and has been widely used to automate proofs in some proof assistants such as Coq [54] and Agda [12]. For example, we can prove an integer equation $(a - b) + (b - a) = 0$ as follows.

1. We obtain a term $e := \mathrm{Add}(\mathrm{Sub}(\mathrm{X}_0, \mathrm{X}_1), \mathrm{Sub}(\mathrm{X}_1, \mathrm{X}_0))$ from the LHS of the equation, where $\mathrm{Add}, \mathrm{Sub} : \mathrm{E} \to \mathrm{E} \to \mathrm{E}$ and $\mathrm{X} : \mathbb{N} \to \mathrm{E}$ are constructors of an inductive type E describing the syntax. This step is called *reification* (also called metaification or quotation).

2. We normalize e to a formal sum $0\mathrm{X}_0 + 0\mathrm{X}_1$ and check that all its coefficients are zero. This decision procedure is implemented and performed inside the proof assistant, and its validity is justified by a correctness lemma.

This process (detailed in Section 2.3 and 2.4) applies to any equation over an Abelian group, and this proof scheme can be adapted to other mathematical structures, e.g., commutative rings [11, 31], fields [60], lattices [34], and Kleene algebras [14].

Unfortunately, existing implementations of this proof methodology are known not to cooperate with *packed classes* very well [27, 33]. The packed classes discipline [26] is a methodology to define mathematical structures in dependent type theory, which allows for the sharing of vocabulary (definitions and lemmas) across the inheritance hierarchy of structures as well as multiple inheritance (Section 2.1 and 2.2). This methodology is used in the MathComp library [39] for Coq extensively, to provide more than 70 mathematical structures such as finite groups, rings, fields, as well as their homomorphisms.

The source of the incompatibility between proof by large-scale reflection and packed classes is twofold. Firstly, packed classes require the proof tools (e.g., the `rewrite` tactic) to compare overloaded operators (e.g., the multiplication of rings) modulo conversion to enable the sharing of vocabulary. This conversion is another kind of computational reflection, so-called *small-scale reflection.* Secondly, in most of the existing tactics based on large-scale reflection, their reification procedures recognize operators purely syntactically and do not take conversion into account. We briefly review an existing approach to matching modulo conversion, called the *keyed matching* discipline [28], used in the SSReflect plugin [58] and the Lean theorem prover [41, 42], and propose a reification scheme based on keyed matching to address this shortcoming (Section 3).

Another issue is that extending the above reflection scheme to support homomorphisms, whose domain and codomain types may differ, requires a more involved data type describing the syntax, another decision procedure, and correctness proof. In this paper, instead of redefining the syntax and the decision procedure, we propose a solution based on two reflection steps. The first step, which we call *preprocessing*, pushes down homomorphisms in the input terms to leaves using the structure preservation laws, e.g., $f(x + y) = f(x) + f(y)$. Although preprocessing requires a heterogeneous syntax that can express a term that has subterms of different types, it remains quite simple since we do not have to replace variables with numbers in preprocessing as in X_n above. In the second step, we apply the reflexive decision procedure that uses a homogeneous syntax, as explained at the beginning of this section. While some generic goal preprocessing methodologies [7, 9, 36] have been implemented as standalone tactics, the characteristic of our approach is that those two steps are strongly tied and the preprocessor is not intended to be called solely. This approach allows us to avoid performing reification twice and thus has a performance advantage. Moreover, the preprocessing step allows us to adapt an existing reflexive tactic to operators not directly supported by its syntax (e.g., opposite which can be expressed as a combination of zero and subtraction) without modifying the existing syntax, procedures, and correctness proofs for the second step (Section 4).

As an application of our methodology, we adapt the `ring` and `field` tactics [31, 60] of Coq to the commutative ring and field structures of MathComp, with support for homomorphisms and some operators that cannot be directly described by the provided syntax (Section 5.1). Furthermore, we apply the resulting tactics to the formal proof of Apéry's theorem [16, 17, 37] (Section 5.3).

▶ **Theorem 1** (Apéry's theorem [3, 62])**.** *The following constant, the evaluation of the Riemann zeta function at 3, is irrational:*

$$\zeta(3) = \sum_{i=1}^{\infty} \frac{1}{i^3}.$$

For this purpose, we also reimplemented a technique [17, Section 4.3][37, Section 2.4] to automatically prove proof obligations generated by the `field` tactic using the `lia` (linear integer arithmetic) tactic [6, 59] of Coq (Section 5.2). This reimplementation is extensible by

declaring canonical structure instances and supports a broader range of problems, thanks to the approach of Gonthier et al. [29] to use canonical structures (Section 2.1) for proof automation. As a result, the lines of code in the formal proof of Apéry's theorem have been reduced by 8% and the time required for proof checking has been reduced by 27%.

Our reification procedures are written in Coq-Elpi [53] (Section 2.4). Elpi [24, 45] is a dialect of λProlog [40], a higher-order logic programming language. The Coq-Elpi plugin lets us write Coq commands and tactics in Elpi, and provides a higher-order abstract syntax (HOAS) [44] embedding of Coq terms in Elpi, to manipulate syntax trees with binders in a comfortable way.

2   Background

This preliminary section briefly reviews the main ingredients of this paper, namely, canonical structures (Section 2.1), the hierarchy of mathematical structures in MathComp (Section 2.2), large-scale reflection (Section 2.3), and reification in Coq-Elpi (Section 2.4).

2.1   Canonical structures

Canonical structures [38, 47, 57] make it possible to implement ad-hoc inference mechanisms in Coq by giving a particular form of hints [4] to the unification engine [63]. An interface to trigger such an inference is expressed as a record. For example, a record type declaration

```
1  Structure eqType := { eq_sort : Type; eq_op : eq_sort -> eq_sort -> bool }.
```

represents a type (`eq_sort`) equipped with a comparison function (`eq_op`). At the same time, `eqType` is an interface to relate a type to its canonical comparison function. **Structure** is just a synonym of **Record**, but we reserve the former for interfaces for canonical structure resolution. A hint can be given as a record instance. For example, an instance

```
1  Canonical nat_eqType : eqType := {| eq_sort := nat; eq_op := eqn |}.
```

allows us to type check `@eq_op _ 0%N 1%N`, where `0%N` and `1%N` are Peano natural numbers of type `nat` and `eqn` is the comparison function of type `nat -> nat -> bool`. Since `eq_op` is a record projection of type (`forall e : eqType, eq_sort e -> ...`), supplying `0%N` as the second argument of `eq_op` requires solving a type equation `eq_sort ?e ≜ nat` to type check, where `?e` is a unification variable of type `eqType`. For a **Canonical** declaration, the system synthesizes a unification hint between the projections (`eq_sort` and `eq_op`) and the head symbols of the fields (`nat` and `eqn`), respectively. Therefore, the above equation is solved by instantiating `?e` with `nat_eqType`.

Additionally, declaring the `eq_sort` projection as an implicit coercion [46, 47, 56] allows us to use `T : eqType` in the context that expects a term of type `Type`, so that one may write `x : T` rather than `x : eq_sort T`.

```
1  Coercion eq_sort : eqType >-> Sortclass.
```

2.2   The hierarchy of mathematical structures in MathComp

We summarize some mathematical structures provided by the MathComp library below and illustrate the inheritance hierarchy they form in Figure 1. Note that `comRingType` and later structures are used only in Section 5. Each structure is defined as a record bundling a `Type` with operators and axioms as in `eqType` of Section 2.1. More details on the structures and their operators can be found in [18, Chapter and 4].
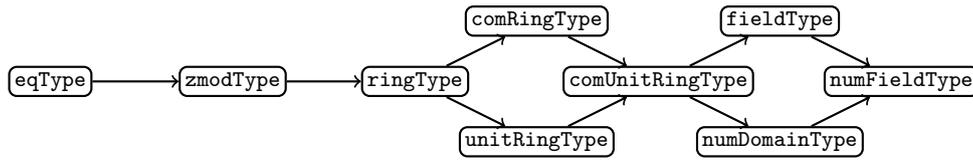
■ **Figure 1** An excerpt of the hierarchy of mathematical structures in MathComp, where an arrow from `aType` to `bType` means that `bType` inherits from `aType`, e.g., `ringType` inherits from `zmodType`.

- `T : eqType` is a type whose propositional equality is decidable. The `eqType` record in Section 2.1 is a simplified version of this structure. For any x and y of type T, `x == y` (:= `eq_op x y`) tests if x is equal to y. Its negation can be expressed as `x != y`.
- `V : zmodType` is a ℤ-module (additive Abelian group). For any x and y of type V, `x + y` (:= `GRing.add x y`), `- x` (:= `GRing.opp x`), and `0` (:= `GRing.zero V`) denotes the sum of x and y, the opposite of x, and zero, respectively.
- `R : ringType` is a ring. For any x and y of type R, `x * y` (:= `GRing.mul x y`) and `1` (:= `GRing.one R`) denotes the product of x and y, and one, respectively.
- `R : comRingType` is a commutative ring.
- `R : unitRingType` is a ring structure with computable inverses. For any x of type R, `x^-1` (:= `GRing.inv x`) denotes the multiplicative inverse of x, which is equal to x itself if x is not a unit, i.e., has no multiplicative inverse.
- `R : comUnitRingType` is a commutative ring with computable inverses.
- `F : fieldType` is a field.
- `R : numDomainType` is a partially ordered integral domain.
- `F : numFieldType` is a partially ordered field.

where "E_1 (:= E_2)" means that E_1 is a notation [55] for E_2, and they are syntactically equal. Each operator above takes a structure instance as its first argument, which is implicit except for `GRing.zero` and `GRing.one`.

These structures are defined by following packed classes, advocated by Garillot et al. [26] and also detailed in [1, 25, 39, 48]. For example, the `ringType` structure is defined as follows.

```
1   (* in Module GRing: *)
2   Module Ring.
3
4   Record mixin_of (R : zmodType) : Type :=
5     Mixin { one : R; mul : R -> R -> R; ... (* properties of one and mul *) }.
6
7   Record class_of (R : Type) : Type :=
8     Class { base : Zmodule.class_of R; mixin : mixin_of (Zmodule.Pack base) }.
9
10  Structure type : Type := Pack { sort : Type; class : class_of sort }.
11
12  Definition zmodType (cT : type) : zmodType :=
13    @Zmodule.Pack (sort cT) (base (class cT)).
14
15  End Ring.
16  Notation ringType := Ring.type.
```

The `GRing.Ring` module serves as a namespace qualifying the definitions inside the module, which are internals to define the `ringType` structure. Each structure has such a module, e.g., `GRing.Zmodule` is for `zmodType`. The structure is divided into three kinds of records: mixin (Line 4), class (Line 7), and structure (Line 10). The mixin record gathers operators and

axioms newly introduced by the structure, e.g., the multiplication, multiplicative identity, and their properties are required to define rings by extending $\mathbb{Z}$-modules. The class record assembles the mixins of the superclasses. The structure record is the actual interface of the structure that bundles a `Type` with its class instance.

GRing.Ring.zmodType is an explicit subtyping function that takes a `ringType` and returns its underlying `zmodType`, which can be made implicit by declaring it as a coercion.

```
1  Coercion Ring.sort : Ring.type >-> Sortclass.
2  Coercion Ring.zmodType : Ring.type >-> Zmodule.type.
```

Furthermore, declaring this subtyping function as a canonical instance allows us to write a term that mixes $\mathbb{Z}$-module and ring operators, e.g., `0 + 1`, by solving type equation of the form GRing.Zmodule.sort ?V $\triangleq$ GRing.Ring.sort ?R. In general, solving an equation GRing.Ring.sort ?R $\triangleq$ T gives us the ring instance ?R of type T.

```
1  Canonical Ring.zmodType.
```

The ring operators are defined by lifting the projections of the mixin record to the structure record, as follows.

```
1  Definition one (R : ringType) : R := Ring.one (Ring.mixin (Ring.class R)).
2  Definition mul (R : ringType) : R -> R -> R := Ring.mul (Ring.mixin (Ring.class R)).
```

Packed classes can also express the hierarchy of morphisms. For example, the MathComp library provides the structure {additive U -> V} of additive functions ($\mathbb{Z}$-module homomorphisms) from U to V. Its record projection GRing.Additive.apply returns a function of type U -> V and is used for triggering instance resolution (e.g., Section 4.1) in the same way as GRing.Ring.sort above. Similarly, there is a structure of ring homomorphisms {rmorphism R -> S} which inherits from additive functions.

2.3 Large-scale reflection

This section demonstrates how to prove $\mathbb{Z}$-module equations by reflection. Firstly, we define the data type describing the syntax as follows:

```
1  Inductive AGExpr : Type :=
2    | AGX : nat -> AGExpr
3    | AGO : AGExpr                        (* zero *)
4    | AGOpp : AGExpr -> AGExpr            (* opposite *)
5    | AGAdd : AGExpr -> AGExpr -> AGExpr. (* addition *)
```

where `AGX j` means j^{th} variable. This inductive data type allows us to write a Coq function manipulating the syntax. For example, we can interpret a syntax tree as follows:

```
1  Fixpoint AGeval (V : Type) (zero : V) (opp : V -> V) (add : V -> V -> V)
2                  (varmap : list V) (e : AGExpr) : V :=
3    match e with
4      | AGX j => nth zero varmap j
5      | AGO => zero
6      | AGOpp e1 => opp (AGeval zero opp add varmap e1)
7      | AGAdd e1 e2 =>
8        add (AGeval zero opp add varmap e1) (AGeval zero opp add varmap e2)
9    end.
```

where the first four arguments are the carrier type and operators of a $\mathbb{Z}$-module, and `varmap` is a list whose j^{th} item gives the interpretation of the jth variable. Such an object representing variable assignments is called a *variable map*.

Similarly, we can define a function `AGnorm` of type `AGExpr -> list int` that normalizes a syntax tree to a list of integers representing a formal sum, e.g., $[1; -2]$ represents $X_0 - 2X_1$, where `int` is the type of integers defined in MathComp. Their correctness specialized for the case that `V` is `int` can be stated as follows.

```
1  Lemma int_correct (varmap : list int) (e1 e2 : AGExpr) :
2    (* if all the coefficients of the normal form of e1 - e2 is equal to 0, *)
3    all (fun i => i == zeroz) (AGnorm (AGAdd e1 (AGOpp e2))) = true ->
4    (* e1 and e2 evaluated to integers by AGeval are equal. *)
5    AGeval zeroz oppz addz varmap e1 = AGeval zeroz oppz addz varmap e2.
```

where `zeroz`, `oppz`, and `addz` are $\mathbb{Z}$-module operators for `int`.

Suppose we want to prove a goal

```
1    (x + (- y)) + x = (- y) + (x + x)
```

for some `x y : int` where + and – here mean `addz` and `oppz`, respectively. Thanks to the above reflection lemma, the proof can be done by the following proof term.

```
1  let e1 := AGAdd (AGAdd (AGX 0) (AGOpp (AGX 1))) (AGX 0) in
2  let e2 := AGAdd (AGOpp (AGX 1)) (AGAdd (AGX 0) (AGX 0)) in
3  @int_correct [:: x; y] e1 e2 erefl.
```

Here we used computational reflection twice. Firstly, `e1` and `e2` are the reified terms representing the LHS and RHS of the goal. These terms interpreted by (`AGeval ... [:: x; y]`) are convertible to the LHS and RHS, respectively. This conversion is triggered by applying the proof term (`@int_correct ...`) to the goal. Secondly, the nullity conditions (`all ... = true`) required by the reflection lemma `int_correct` is checked by reducing its LHS to `true`. This conversion is triggered by supplying `erefl` as the last argument of `int_correct` because `erefl` is the reflexivity proof of type (`forall x, x = x`) where the argument `x` can be left implicit. In the former case, unfolding too many constants may lead to performance issues, and conversion should be performed carefully. In the latter case, we can simply reduce the LHS to `true`, and this is the case where optimized reduction procedures such as `vm_compute` [30] and `native_compute` [10] can be useful.

2.4   Implementing reification in Coq-Elpi

To turn the above method into an automated proof tool, the reified terms and the variable map must automatically be obtained from the goal. Since we cannot pattern match on the operators such as `oppz` in the object level, this reification has to be done in the meta level.

In this section, we implement reification in Coq-Elpi. An example of an Elpi program follows.

```
1  pred mem o:list term, o:term, o:term.
2  mem [X|_]  X {{ O }}       :- !.
3  mem [_|XS] X {{ S lp:N }} :- !, mem XS X N.
```

In this code, we define a *predicate* `mem`. Line 1 is the type signature of `mem`, meaning that `mem` has three arguments of type `list term`, `term`, and `term`, respectively, where `term` is the type of Coq terms embedded in Elpi. Line 2 and 3 are two *rules* that define the meaning of `mem`. Capital identifiers such as `X`, `XS`, and `N` are unification variables. The syntax `[X|XS]` is a

cons cell of lists whose head and tail are `X` and `XS`, respectively. The syntaxes `{{ ... }}` and `lp:` are the quotation from Elpi to Coq and the antiquotation from Coq to Elpi, respectively. Therefore, these two rules are equivalent to the following:

```
1  mem [X|_]  X (global (indc «O»))         :- !.
2  mem [_|XS] X (app [global (indc «S»), N]) :- !, mem XS X N.
```

where `app` of type `list term -> term` is a constructor of `term` meaning an n-ary function application of Coq, and `global (indc _)` means a constructor of Coq.[1]

Actually, the proposition `mem XS X N` asserts that the $N + 1^{\text{th}}$ element of `XS` is `X`, where `N` is a Coq term of type `nat`. Let us consider an example `mem [Y, Z] Z M`, where `Y` and `Z` are distinct Coq terms and `M` remains unknown. The LHS of the first rule requires that the head of `XS` is `X`, but this does not apply to our example. Thus, it attempts matching with the second rule by solving equations `[_|XS] = [Y, Z]`, `X = Z`, and `{{ S lp:N }} = M`. Then we get `XS = [Z]` from the first equation, and proceed to execute its RHS (`!, mem [Z] Z N`), which is the conjunction of the cut (`!`) operator and `mem [Z] Z N`. The cut operator prevents backtracking, i.e., trying other rules of `mem` when the later items of the conjunction fails. Since `mem [Z] Z N` matches with the first rule, `N` is instantiated with `{{ O }}`. In the end, our example `mem [Y, Z] Z M` succeeds with `S O` substituted to the variable `M`. Indeed, `Z` is the second element of `[Y, Z]`.

If the first argument `XS` is an open-ended list `[X0, ..., XN | XS']` where `XS'` remains unknown, and the given item `X` is none of the known elements, `mem XS X _` instantiates `XS'` with `[X | _]` and `X` becomes the $N + 2^{\text{th}}$ element of `XS`.

We implement reification as a predicate `quote`, such that `quote In Out VarMap` reifies `In` of type `int` to `Out` of type `AGExpr` and produces an open-ended variable map `VarMap`:

```
1  pred quote i:term, o:term, o:list term.
2  quote {{ zeroz }} {{ AGO }} _ :- !.
3  quote {{ oppz lp:In1 }} {{ AGOpp lp:Out1 }} VarMap :- !,
4    quote In1 Out1 VarMap.
5  quote {{ addz lp:In1 lp:In2 }} {{ AGAdd lp:Out1 lp:Out2 }} VarMap :- !,
6    quote In1 Out1 VarMap, quote In2 Out2 VarMap.
7  quote In {{ AGX lp:N }} VarMap :- !, mem VarMap In N.
```

where `i:` and `o:` stand for input and output, respectively. Marking an argument as input avoids instantiation of that argument. The first three rules of `quote` are just simple syntactic translation rules for the operators. If the input does not match with any of those, it should be treated as a variable by the last rule, which is implemented using the `mem` predicate above.

3 Large-scale reflection for packed classes

Thanks to the techniques reviewed in Section 2.3 and 2.4, we can implement a tactic `int_zmodule` for solving any integer equation that holds for any $\mathbb{Z}$-module. However, its generalization `poly_zmodule` to arbitrary $\mathbb{Z}$-modules, declared as instances of the `zmodType` structure, is actually not trivial. First, we describe a naive implementation that fails and analyze the source of the failure in Section 3.1. Then, we propose a solution to this issue based on the keyed matching discipline [28] in Section 3.2.

---

[1] Note that the actual Elpi syntax does not support «...» in input. This is just for illustration purposes.

3.1   Purely syntactic reification does not work for packed classes

We first generalize the correctness and reflection lemmas `int_correct` to any ℤ-module:

```
1  Lemma AG_norm_subst (V : zmodType) (varmap : list V) (e : AGExpr) :
2    AGsubst 0 -%R +%R varmap (AGnorm e) = AGeval 0 -%R +%R varmap e.
3
4  Lemma AG_correct (V : zmodType) (varmap : list V) (e1 e2 : AGExpr) :
5    all (fun i => i == 0) (AGnorm (AGAdd e1 (AGOpp e2))) = true ->
6    AGeval 0 -%R +%R varmap e1 = AGeval 0 -%R +%R varmap e2.
```

where `AG_norm_subst` is the key lemma to prove `AG_correct`, `AGsubst` is the function to substitute a variable map to a formal sum, and `-%R` and `+%R` are 0-ary notations for `GRing.opp` and `GRing.add` implicitly applied to `V`, respectively.

To reimplement the `quote` predicate, we add a new argument `V` which is the `zmodType` instance for the type of the input term, and replace operators `zeroz`, `oppz`, and `addz` with `@GRing.zero V`, `@GRing.opp V`, and `@GRing.add V`, respectively.

```
1  pred quote i:term, i:term, o:term, o:list term.
2  quote V {{ @GRing.zero lp:V }} {{ AGO }} _ :- !.
3  quote V {{ @GRing.opp lp:V lp:In1 }} {{ AGOpp lp:Out1 }} VarMap :- !,
4    quote V In1 Out1 VarMap.
5  quote V {{ @GRing.add lp:V lp:In1 lp:In2 }} {{ AGAdd lp:Out1 lp:Out2 }} VarMap :- !,
6    quote V In1 Out1 VarMap, quote V In2 Out2 VarMap.
7  quote _ In {{ AGX lp:N }} VarMap :- !, mem VarMap In N.
```

However, this `quote` predicate fails to reify at least one addition operator in the goal:

```
1  forall x : int, x + 1 = 1 + x.
```

Let us take a closer look at it by **Set Printing All**:

```
1  forall x : int,
2  @eq (GRing.Zmodule.sort int_ZmodType)
3    (@GRing.add int_ZmodType x (GRing.one int_Ring))
4    (@GRing.add (GRing.Ring.zmodType int_Ring) (GRing.one int_Ring) x)
```

where `int_ZmodType` and `int_Ring` are the canonical `zmodType` and `ringType` instances of `int`, respectively.

The root of the issue is that the two occurrences of `GRing.add` take syntactically different `zmodType` instances as highlighted. The former instance is inferred from the type of `x`, by solving the type equation `GRing.Zmodule.sort ?V ≜ int`. The latter instance is inferred from the type of `GRing.one ?R` where `?R` is eventually instantiated with `int_Ring`, by solving the type equation `GRing.Zmodule.sort ?V ≜ GRing.Ring.sort ?R` whose solution is `?V := GRing.Ring.zmodType ?R`. The `quote` predicate above requires that all the `zmodType` instances occurring as the first argument of the operators are syntactically equal to each other. However, the above goal does not respect this restriction. In the presence of the inheritance mechanism of packed classes, such syntactically different instances for the same type and structure coexist [1, Section 3.1][26, Section 2.4][48, Section 3], and canonical structure resolution may infer them simultaneously. Nevertheless, definitional equality of those instances is ensured by *forgetful inheritance* [1], that is, the practice of implementing inheritance and subtyping functions by record inclusion and erasure of some record fields, respectively.

3.2 Reification by small-scale reflection

Reification recognizing operators by conversion or unification rather than purely syntactic matching would address the above issue. However, using full unification for term matching, e.g., triggering unification `@GRing.opp V ?t'` ≜ `t` to check if `t` is the opposite of an unknown term `?t'`, can make reification too costly. Thus, we propose a solution that mixes syntactic matching and conversion as in the keyed matching discipline [28]. The idea of keyed matching is to find a subterm that matches with a pattern (`f t1 ... tn`) by attempting the matching operation only on subterms of the form (`f t1' ... tn'`). While the head constant (the key) `f` has to be the same constant,[2] its arguments can be compared by conversion or unification.

In our case, the keys are the $\mathbb{Z}$-module operators `GRing.zero`, `GRing.opp`, and `GRing.add`. The `quote` predicate can be reimplemented as follows:

```
1  pred quote i:term, i:term, o:term, o:list term.
2  quote V {{ @GRing.zero lp:V' }} {{ AG0 }} _ :- coq.unify-eq V V' ok, !.
3  quote V {{ @GRing.opp lp:V' lp:In1 }} {{ AGOpp lp:Out1 }} VarMap :-
4    coq.unify-eq V V' ok, !, quote V In1 Out1 VarMap.
5  quote V {{ @GRing.add lp:V' lp:In1 lp:In2 }} {{ AGAdd lp:Out1 lp:Out2 }} VarMap :-
6    coq.unify-eq V V' ok, !, quote V In1 Out1 VarMap, quote V In2 Out2 VarMap.
7  quote _ In {{ AGX lp:N }} VarMap :- !, mem VarMap In N.
```

where `coq.unify-eq V V' ok` asserts that `V` unifies with `V'`. Since the first argument `V` and the input term do not have any unification variable under normal use of `quote`, this unification problem falls in a conversion problem that is generally easier and less costly to solve than unification. For example, the second rule of `quote` (Line 3) does not require `V'` in the input term `@GRing.opp V' In1` to be syntactically equal to the first argument `V`, but it compares `V'` with `V` by conversion after syntactic matching of the opposite operator `GRing.opp`. Since this conversion is a part of term matching, the cut operator to prevent backtracking comes after conversion.

The `zmodType` instance to use as the first argument of `quote` can be obtained by canonical structure resolution. This inference is implemented as follows.

```
1  pred solve i:goal, o:list sealed-goal.
2  solve (goal _ _ {{ @eq lp:Ty lp:T1 lp:T2 }} _ _ as G) GS :-
3    std.assert-ok! (coq.unify-eq {{ GRing.Zmodule.sort lp:V }} Ty)
4                   "Cannot find a declared Z-module", !,
5    quote V T1 ZE1 VarMap, !, quote V T2 ZE2 VarMap, !,
6    ...
```

The `solve` predicate is the entry point of a tactic in Coq-Elpi. The above rule matches the goal proposition with a pattern `T1 = T2` where `T1` and `T2` have type `Ty` (Line 2), triggers unification `GRing.Zmodule.sort V` ≜ `Ty` to find the canonical `zmodType` instance `V` of `Ty` (Line 3), and then reifies `T1` to `ZE1` and `T2` to `ZE2` using `V` obtained in the second step (Line 5). Note that if unification by `coq.unify-eq` fails, its third argument of type `diagnostic` carries the error message. The `std.assert-ok!` predicate of Line 3 asserts that unification given as the first argument succeeds, but if it fails, it prints the carried error message with the string given as the second argument.

---

[2] In the actual implementation of keyed matching in the SSReflect plugin, there are some exceptions such that a projection of a canonical structure can match with its canonical instances. See [28, Section 3.1] for details.

4   Extending the syntax with homomorphisms and more operators

In this section, we implement a new tactic `morph_zmodule`, that extends the syntax supported by `poly_zmodule` with $\mathbb{Z}$-module homomorphisms (Section 4.1) and subtraction (Section 4.2) which is not directly supported the syntax `AGExpr`. These extensions are achieved by adding another layer of reflection which we call *preprocessing*. This twofold reflection scheme allows us to reuse the syntax `AGExpr`, interpretation and normalization procedures `AGeval` and `AGnorm`, and the reflection lemma `AG_correct` presented in Section 2 and 3 as is.

4.1   Homomorphisms

Firstly, we define another inductive type describing the syntax involving homomorphisms.

```
1  Implicit Types (U V : zmodType).
2
3  Inductive MExpr : zmodType -> Type :=
4    | MX V : V -> MExpr V
5    | MO V : MExpr V
6    | MOpp V : MExpr V -> MExpr V
7    | MAdd V : MExpr V -> MExpr V -> MExpr V
8    | MMorph U V : {additive U -> V} -> MExpr U -> MExpr V.
```

The main difference of this type compared with `AGExpr` is that: `MExpr` (Line 3) is parameterized by a `zmodType` instance `V`, the constructor `MX` (Line 4) representing a variable takes a term of type `V` instead of an index of type `nat`, and the constructor `MMorph` (Line 8), representing a homomorphism application, allows for changing the parameter `V`. Therefore, one shall interpret a reified term of this type without a variable map provided.

```
1  Fixpoint Meval V (e : MExpr V) : V :=
2    match e with
3      | MX _ x => x
4      | MO _ => 0
5      | MOpp _ e1 => - Meval e1
6      | MAdd _ e1 e2 => Meval e1 + Meval e2
7      | MMorph _ _ f e1 => f (Meval e1)
8    end.
```

The normalization procedure we need for `MExpr` is just pushing down homomorphisms appearing as the `MMorph` constructor to the leaves of the syntax tree:

```
1  Fixpoint Mnorm U V (f : {additive U -> V}) (e : MExpr U) : V :=
2    match e in MExpr U return {additive U -> V} -> V with
3      | MX _ x => fun f => f x
4      | MO _ => fun _ => 0
5      | MOpp _ e1 => fun f => - Mnorm f e1
6      | MAdd _ e1 e2 => fun f => Mnorm f e1 + Mnorm f e2
7      | MMorph _ _ g e1 => fun f => Mnorm [additive of f \o g] e1
8    end f.
```

where the third argument (`f : {additive U -> V}`) accumulates homomorphisms applied to `e`. Therefore, the case for `e := MMorph _ _ g e1` (Line 7) constructs a homomorphism `[additive of f \o g]` that is the function composition of `f` and `g`, and passes it to the recursive call for normalizing `e1`. On the other hand, the case for `e := MX _ x` (Line 3) applies `f` to the variable `x`. Since dependent pattern matching on `e : MExpr U` forces instantiation of `U` in type checking of each clause, defining `Mnorm` that type checks requires the so-called *convoy pattern* [15] to propagate this instantiation to the type of `f`.

Thanks to the structure preservation laws of homomorphisms, a result of normalization `Mnorm f e` should be equal to `f` applied to `Meval e`. That is to say, the following correctness lemma holds:

```
1  Lemma M_correct V (e : MExpr V) : Meval e = Mnorm [additive of idfun] e.
```

where `[additive of idfun]` is the identity homomorphism.

The reification procedure for the `morph_zmodule` tactic should take an input term `In` of type `V : zmodType`, and obtain a variable map `varmap` and two reified terms `OutM` and `Out` of types `MExpr V` and `AGExpr`, respectively. For any such `Coq` terms, the following chain of equations should hold to justify the completeness of the tactic:

$$
\begin{array}{llr}
\texttt{In} \equiv \texttt{Meval OutM} & & \text{(a meta property)} \\
\quad = \texttt{Mnorm [additive of idfun] OutM} & & \text{(Lemma M\_correct)} \\
\quad \equiv \texttt{AGeval ... varmap Out} & & \text{(a meta property)} \\
\quad = \texttt{AGsubst ... varmap (AGnorm Out)} & & \text{(Lemma AG\_norm\_subst)}
\end{array}
$$

where ≡ and = respectively mean definitional equality and propositional equality. Although these meta properties of reification cannot be proved inside `Coq`, the kernel of `Coq` will check them for every invocation of the tactic, as explained in Section 2.3.

Considering the above requirements, reification can be reimplemented as follows.

```
1  pred quote i:term, i:(term -> term), i:term, o:term, o:term, o:list term.
2  quote V _ {{ @GRing.zero lp:V' }} {{ MO lp:V }} {{ AGO }} _ :-
3    coq.unify-eq V V' ok, !.
4  quote V F {{ @GRing.opp lp:V' lp:In1 }}
5        {{ @MOpp lp:V lp:OutM1 }} {{ AGOpp lp:Out1 }} VarMap :-
6    coq.unify-eq V V' ok, !, quote V F In1 OutM1 Out1 VarMap.
7  quote V F {{ @GRing.add lp:V' lp:In1 lp:In2 }}
8        {{ @MAdd lp:V lp:OutM1 lp:OutM2 }} {{ AGAdd lp:Out1 lp:Out2 }} VarMap :-
9    coq.unify-eq V V' ok, !,
10   quote V F In1 OutM1 Out1 VarMap, quote V F In2 OutM2 Out2 VarMap.
11 quote V F In {{ @MMorph lp:U lp:V lp:G lp:OutM }} Out VarMap :-
12   coq.unify-eq {{ @GRing.Additive.apply lp:U lp:V lp:Ph lp:G lp:In1 }} In ok, !,
13   quote U (x\ F {{ @GRing.Additive.apply lp:U lp:V lp:Ph lp:G lp:x }})
14        In1 OutM Out VarMap.
15 quote V F In {{ @MX lp:V lp:In }} {{ AGX lp:N }} VarMap :- !,
16   mem VarMap (F In) N.
```

Let the six arguments of the new `quote` predicate be `V`, `F`, `In`, `OutM`, `Out`, and `VarMap`. The first three arguments are input: `V` is a `zmodType` instance, `F` is a homomorphism from `V` to another `zmodType` instance, and `In` is the input term of type `V`. Then, the last three arguments are output: `OutM` and `Out` are the reified terms of types `MExpr V` and `AGExpr`, respectively, and `VarMap` is the variable map. The second argument `F` is required to make recursion of `quote` work and accumulates homomorphisms as in the third argument `f` of `Mnorm`. Note that `F` is represented as an `Elpi` function from `term` to `term`, which lets us compose functions without leaving a beta redex in the `Coq` level. While the first reified term `OutM` exactly corresponds to `In`, the second reified term `Out` and the variable map corresponds to `F In`.

The most crucial part of the new `quote` predicate is its fourth rule (Line 11), which handles the case that the input `In` is a homomorphism application. It first triggers unification `@GRing.Additive.apply U V _ G In1 ≜ In` to decompose the input into the homomorphism instance `G` and its argument `In1`. Then, since `G` is a homomorphism from `U` to `V`,

it invokes the recursive call of `quote` on `In1` with `U` as the first argument (the `zmodType` instance) and the composition of `F` and `G` as the second argument (the homomorphism). This composition is written as `(x\ ...)` which means an abstraction ($\lambda x. \ldots$).

4.2 More operators

Based on our twofold reflection scheme, we can add support for operators not directly supported by the syntax `AGExpr`. For example, let `subr` be an opaque subtraction operator of type (`forall` U : zmodType, U -> U -> U). By opaque we mean that `subr` does not reduce and thus we cannot rely on its definitional behavior, but we can reason about it through a lemma:

```
1  subrE : @subr = (fun (U : zmodType) (x y : U) => x + (- y)).
```

Firstly, we add the following constructor to `MExpr` representing `subr`.

```
1  (* in Inductive MExpr: *)
2    | MSub V : MExpr V -> MExpr V -> MExpr V
```

Then, the interpretation and normalization function have to be adapted to the new definition of `MExpr`, by adding the following cases.

```
1  (* in Fixpoint Meval: *)
2      | MSub _ e1 e2 => subr (Meval e1) (Meval e2)
```

```
1  (* in Fixpoint Mnorm: *)
2      | MSub _ e1 e2 => fun f => Mnorm f e1 + (- Mnorm f e2)
```

The point is that `Meval` interprets `MSub V e1 e2` using `subr`, but `Mnorm` normalizes it using `GRing.add` and `GRing.opp`. Proving the correctness lemma `M_correct` based on these new definitions can be done using the lemma `subrE`. These definitions and the correctness lemma let us replace subexpressions of the form `subr e1 e2` with `e1' + (- e2')` by preprocessing, and make it possible to generate a corresponding reified term of type `AGExpr`. Therefore, an input term of the form `subr _ _` has to be reified to `@MSub _ _ _` and `AGAdd _ (AGOpp _)`, as follows (see Line 2).

```
1  quote V F {{ @subr lp:V' lp:In1 lp:In2 }}
2      {{ @MSub lp:V lp:OutM1 lp:OutM2 }} {{ AGAdd lp:Out1 (AGOpp lp:Out2) }}
3      VarMap :-
4    coq.unify-eq V V' ok, !,
5    quote V F In1 OutM1 Out1 VarMap, quote V F In2 OutM2 Out2 VarMap.
```

In fact, even if an operator can be supported by relying on its definitional behavior, our methodology is sometimes performance-wise better than doing so. For example, `n%:R` (:= 1 *+ n) and `n%:~R` (:= 1 *~ n) are generic embeddings of `n : nat` and `n : int` to a ring, respectively, where `x *+ n` (:= GRing.natmul x n) and `x *~ n` (:= intmul x n) are `n` times addition of `x : V` defined for any `V : zmodType`. For any `n` of type `nat`, `n%:R` and `(Posz n)%:~R` are convertible since the latter unfolds to the former, where `Posz` is a constructor of `int` that embeds `nat` to `int`. Therefore, if a reflexive tactic supports `n%:~R`, it is possible to support `n%:R` by reifying it in the same way as `(Posz n)%:~R`. However, it may lead to performance issues by triggering conversions such as:

```
1  Time Check erefl : (Posz 6 * 6)%:~R = 36%:R :> rat. (* 36.364s *)
```

where `:> rat` means that the LHS and RHS are rational numbers of type `rat`.[3]

---

[3] Note that this particular performance issue reproduces only with MathComp 1.12.0 or earlier.

The source of this inefficiency is that conversion unfolds too many constants. Computations involving rational numbers are particularly inefficient because `rat` is defined as a dependent pair of the numerator and denominator that are coprime [18, Section 4.4.2] and every `rat` operator performs GCD calculation to ensure the canonicity of representations. In our reflection scheme, conversion between `GRing.natmul` and `intmul` can be hidden in preprocessing. It makes conversion performing only a small number of unfolding and thus more efficient.

5 Applications: `ring`, `field`, and the irrationality of $\zeta(3)$

As an application of the methodology presented in Section 3 and 4, we briefly report our effort to adapt the `ring` and `field` tactics [31, 60] of Coq to the commutative ring and field structures of MathComp in Section 5.1. The `field` tactic generates proof obligations describing the non-nullity of the denominators in the given equation. Those conditions can often be simplified to equivalent integer disequations and solved by the `lia` tactic. In Section 5.2, we implement this simplification based on the approach of Gonthier et al. [29] to use canonical structures for proof automation. In Section 5.3, we apply the above proof tools to the formal proof of Apéry's theorem by Chyzak, Mahboubi, and Sibut-Pinote [16, 17, 37] to bring more proof automation. Our `ring` and `field` tactics for MathComp are available as a Coq library called *Algebra Tactics* [49].

5.1 The `ring` and `field` tactics

The `ring` and `field` tactics [31, 60] of Coq respectively solve polynomial and rational equations by computational reflection. Their reflexive decision procedures are based on normalization to the *sparse Horner form* [31], a multivariate, computationally efficient version of the Horner normal form of polynomials.

The following inductive type describes the syntax supported by the `ring` tactic:

```
1  Inductive PExpr (C : Type) : Type :=
2    | PE0 : PExpr C                           (* zero:        GRing.zero *)
3    | PEI : PExpr C                           (* one:         GRing.one  *)
4    | PEc : C -> PExpr C                       (* constant:    _%:~R      *)
5    | PEX : positive -> PExpr C                (* variable                *)
6    | PEadd : PExpr C -> PExpr C -> PExpr C    (* addition:    GRing.add  *)
7    | PEsub : PExpr C -> PExpr C -> PExpr C    (* subtraction: _ - _      *)
8    | PEmul : PExpr C -> PExpr C -> PExpr C    (* multiplication: GRing.mul *)
9    | PEopp : PExpr C -> PExpr C               (* opposite:    GRing.opp  *)
10   | PEpow : PExpr C -> N -> PExpr C.         (* power:       GRing.exp  *)
```

where `C` is the type of coefficients and fixed to the binary integer type `Z` of the Coq standard library in our usage. For each constructor, its meaning and the corresponding operator in MathComp are indicated in the code comment left. Note that `n%:~R` is the generic embedding of `n : int` to a ring explained in Section 4.2, and `x ^+ n (:= GRing.exp x n)` is the nth power of `x` with `n : nat`. There is also `x ^ n (:= exprz x n)` operator, namely, n^{th} power of `x` with `n : int`, which works only for `unitRingType`.

In addition to the above constructs, the `field` tactic supports the following two operators.

```
1  (* in Inductive FExpr: *)
2    | FEinv : FExpr C -> FExpr C               (* inverse:     GRing.inv  *)
3    | FEdiv : FExpr C -> FExpr C -> FExpr C    (* division:    _ / _      *)
```

On top of these syntaxes, we implemented preprocessors to support homomorphisms and more operators such as `GRing.natmul`, `intmul`, and `exprz`. Since rings and fields have poorer structures such as $\mathbb{Z}$-modules, subexpressions of these structures may appear under homomorphism applications. For example, let `f : V -> R` be an additive function whose codomain `R` is a ring, and we want to perform the following equational reasoning in preprocessing:

```
1    f (x *~ (n * m))
2  = f x * (n * m)%:~R
3  = f x * (n%:~R * m%:~R).
```

This example indicates that ring multiplication may appear in a $\mathbb{Z}$-module subexpression of a ring expression, and homomorphisms can be pushed down through it.

Therefore, we defined three inductive types describing the syntax: `NExpr` for expressions of type `nat`, `RExpr` for ring expressions, and `ZMExpr` for $\mathbb{Z}$-module expressions. The latter two are defined as mutually inductive types. `RExpr` contains constructors for field operators and is used for both **ring** and **field** tactics. Since a ring homomorphism can be pushed down through these operators only if the codomain of the homomorphism is a field, we define normalization functions for `RExpr` and `ZMExpr` for each of the **ring** and **field** tactics separately.

5.2 Automating proofs of non-nullity conditions for **field**

The **field** tactic can now solve a goal

```
1  ((n ^+ 2)%:~R - 1) / (n%:~R - 1) = (n%:~R + 1) :> F
```

where `F` is a field. The **field** tactic then generates a proof obligation `n%:~R - 1 != 0 :> F` describing the non-nullity of the denominator in the equation. If `F` is a partially ordered field (`numFieldType`), this obligation can be simplified to `n != 1 :> int` because `_%:~R` for any partially ordered integral domain (`numDomainType`) is injective. The simplified obligation can sometimes be solved by other automated tactics such as **lia** [6, 59], which can solve linear goals over integers. This combination of the **field** and **lia** tactics is extensively used in the formal proof of Apéry's theorem [17, Section 4.3][37, Section 2.4].

Note that the **lia** tactic has another preprocessing tactic, called **zify**, that canonizes the goal by mapping numeric types such as `nat` to the binary integer type `Z`. The **zify** tactic is extensible by declaring type class instances explaining how to map types and operators in preprocessing. We implemented another small library called Mczify [50] to use this feature to support the arithmetic operators of `MathComp` in the **lia** tactic.

In this section, we reimplement this simplification based on the approach of Gonthier et al. [29]. Firstly, we define a canonical structure `zifyRing` that relates a ring expression that is an element of the integer subring (`rval : R`), to the corresponding integer expression (`zval : int`) such that `rval = zval%:~R`.

```
1  Section ZifyRing.
2
3  Variable R : ringType.
4
5  Structure zifyRing :=
6    ZifyRing { rval : R; zval : int; zifyRingE : rval = zval%:~R }.
```

For instance, the `zifyRing` record allows us to relate `0` and `1` of type `R` to `0` and `1` of type `int`, respectively, as follows.

```
1  Canonical zify_zero := @ZifyRing 0 0 (erefl : 0 = 0%:~R).
2  Canonical zify_one  := @ZifyRing 1 1 (erefl : 1 = 1%:~R).
```

Since the integer subring is closed under opposite, `- x = (- n)%:~R` holds if `x = n%:~R`. This implication can be encoded as a canonical instance that takes another instance as an argument, as follows.

```
1  Lemma zify_opp_subproof (e1 : zifyRing) : - rval e1 = (- zval e1)%:~R.
2
3  Canonical zify_opp (e1 : zifyRing) :=
4    @ZifyRing (- rval e1) (- zval e1) (zify_opp_subproof e1).
```

Similarly, the closure properties under `GRing.add`, `GRing.mul`, and `intmul` can be implemented as the following instances.

```
1  Canonical zify_add e1 e2 := @ZifyRing (rval e1 + rval e2) (zval e1 + zval e2) ...
2  Canonical zify_mul e1 e2 := @ZifyRing (rval e1 * rval e2) (zval e1 * zval e2) ...
3  Canonical zify_mulrz e1 n := @ZifyRing (rval e1 *~ n) (zval e1 *~ n) ...
```

In general, solving an equation `rval ?e1 ≜ x` gives us an integer expression `n` and its correctness proof of `x = n%:~R` from a ring expression `x`. Let us consider an example `x := 1 + n%:~R *~ 2`. Solving the equation `rval ?e1 ≜ x` proceeds by instantiating `?e1` with `zify_add ?e2 ?e3` since the head symbol of `x` is `GRing.add`, and then the problem is divided into two sub-problems `rval ?e2 ≜ 1` and `rval ?e3 ≜ n%:~R *~ 2`. Solving the former sub-problem is done by instantiating `?e2` with `zify_one`, solving the latter proceeds by instantiating `?e3` with `zify_mulrz ?e4 2`, and we get another sub-problem `rval ?e4 ≜ n%:~R`. By repeating this recursive process, we eventually get the canonical solution `?e4 := zify_mulrz zify_one n`. The `zval` and `zifyRingE` fields of the solution `?e1 := zify_add zify_one (zify_mulrz (zify_mulrz zify_one n) 2)` give us the integer expression and the proof, respectively.

Reducing a ring (dis)equation to an integer (dis)equation is performed by rewriting the ring equation by the following lemma.

```
1  End ZifyRing.
2
3  Lemma zify_eqb (R : numDomainType) (e1 e2 : zifyRing R) :
4    (rval e1 == rval e2) = (zval e1 == zval e2).
```

For example, combining the above lemma and the `lia` tactic allows us to solve the following goal. We use a small Ltac [22] script to perform this proof automation in practice.

```
1  Goal forall n : int, n%:~R *~ 2 + 1 != 0 :> rat.
2  Proof. move=> n; rewrite zify_eqb /=; lia. Qed.
```

5.3 The irrationality of $\zeta(3)$

This section briefly reports the result of applying the proof tools presented in the previous sections to the formal proof of Apéry's theorem [16, 17, 37]. This proof involves various collections of numbers such as integers, rational numbers `rat`, their real closure `realalg`, algebraic numbers `algC`, and Cauchy sequences. These types are equipped with ring instances except for Cauchy sequences and also with field instances except for integers. Therefore, the embedding functions corresponding to their inclusion, e.g., $\mathbb{Z} \subset \mathbb{Q}$, are ring homomorphisms. Since the type of integers `int` of `MathComp` is defined based on Peano natural numbers `nat`, it is well suited for proofs but prevents us from performing computation involving large integer constants in a reasonable time. Therefore, this proof also uses the binary representation of

■ **Table 1** Performance comparison of the `rat_field` tactic and our `field` tactic in `ops_for_b.v` of the formal proof of Apéry's theorem, which proves a recurrence equation satisfied by a sequence called b [17, Section 4]. The size of a problem is the number of constructors of reified terms of type `FExpr`. Note that Problem #1 has many relatively large coefficients greater than 10^{12}. Therefore, the ratio of time spent for its normalization is higher than those of the other problems, and thus its improvement in execution time is relatively minor.

Lemma	# Problem	Size	`rat_field` Time (s)	`field` Time (s)
`P_eq_Delta_Q`	1	14,690	91.807	87.013
`recAperyB`	2	8,407	4.170	2.068
`recAperyB`	3	113,657	39.676	26.963

integers `Z` for computation purposes and defines a function that embeds `Z` to rational numbers `rat_of_Z : Z -> rat`. This embedding function is made opaque to prevent computing in `rat`.

We managed to replace two tactics in this proof with our tools: `rat_field` adapting the `field` tactic to MathComp, and `goal_to_lia` implementing the reduction of Section 5.2, both of which are implemented in Ltac and specific to rational numbers `rat`. This replacement is done by adding the support for `Z` constants and operators to our `field` tactic and by making `rat_of_Z` canonically a ring homomorphism. That is to say, we did not have to implement any treatment specific to this proof to our tools, since supporting large integer constants is considered to be of general interest. Moreover, our `ring` and `field` tactics can reason about any ring and field instances and ring homomorphisms. Thus, they can solve a broader range of subgoals, and some manual labor before or after invoking them, e.g., tweaking ring homomorphisms, has been handed off to our tools.

Our tools not only automate more proofs but also, directly and indirectly, make proofs more concise and faster to check. To give some figures, Table 1 summarizes the performance of the invocations of `rat_field` and `field` that take more than 1 second in the proof. In those cases, `field` is consistently faster than `rat_field`. Moreover, by extensively refactoring proofs using our tools, we could reduce 442 lines of specifications and proofs out of 5829 lines excluding code for proof automation, and checking the entire proof became 27% faster (6 min 52 s) than before (9 min 23 s).

On the other hand, we still see some room for improvement in this refactoring work. For example, our `field` tactic cannot directly solve an equation that has rational exponents, e.g., $x^{\frac{3}{2}}$, or variables in exponents [5], e.g., $x^{n+m} = x^n x^m$. However, they require reimplementing reflexive decision procedures and are pretty orthogonal to the present work, except that it might be possible to implement incomplete support for the latter case in preprocessing.

6   Conclusion

We proposed a methodology for building reflexive tactics and their concrete implementations in Coq-Elpi that cooperate with algebraic structures (Section 3) and their homomorphisms (Section 4.1) represented by packed classes. The issue we solved in Section 3 is not specific to packed classes, as the issue solved by forgetful inheritance [1] also appears in semi-bundled [61, Section 4.1] type classes [51]. On the other hand, purely syntactic reification works fine with unbundled [52] type classes, where operators appear as parameters of interfaces, as is the case in [13, 14]. However, this approach does not scale up to larger hierarchies, e.g., as noted in [14, Section 6.1]. Reification by small-scale reflection (Section 3.2) has also been adopted

in the `zify` tactic [8], which is the key ingredient to enable support for overloaded arithmetic operators of MathComp in the `lia` tactic (Section 5.2). Reification by small-scale reflection is also applicable to reification by parametricity [32], although it does not deal with variable maps and thus does not fit our purpose. Such implementation can be done by using the `ssrpattern` tactic in place of the `pattern` tactic, but it may not preserve the efficiency of reification by parametricity.

We argue that Coq-Elpi turned out to be a practical tool to implement our methodology, and in particular, provides features that made our reification procedures concise, although we could reimplement our tactics with other meta-languages such as OCaml, Ltac [22], Ltac2 [43], and Mtac2 [35]. For example, the cut operator, which is unavailable in type classes [52, Section 9] and Ltac, offers a pretty intuitive way to control backtracking and is a key to achieve efficiency. Quotation and antiquotation allow us to embed Coq terms with holes in our Elpi code in a readable way. Moreover, the results in Section 5.3 show that our resulting tactics run in reasonable times.

Our twofold reflection scheme and its preprocessing step allow us to adapt an existing reflexive tactic to homomorphisms (Section 4.1) and new operators (Section 4.2) without reimplementing the whole tactic. While other preprocessing tools such as `zify`, `ppsimpl` [7], and `trakt` [9] have been implemented as standalone tactics, our approach avoids performing reification twice by binding the preprocessor tightly with the reflexive decision procedure and thus has a performance advantage. This idea has been mentioned in [7, Section 3]:

> As future work, we consider binding `ppsimpl` more tightly with other reflective tactic such as `ring` or `lia` thus avoiding to perform a reification twice.

On the other hand, the downside of our approach is that it is less modular and does not let users extend an existing preprocessor with new rules. Since Coq-Elpi provides the abilities to generate inductive data types, Coq constants, and Elpi rules, we could improve this situation by writing an Elpi program that produces a reflexive preprocessor and reification rules from their high-level descriptions. Furthermore, we could integrate such an enhancement to Hierarchy Builder [21] to utilize metadata about the hierarchy of structures in reification.

As an application of our methodology, we adapted the `ring` and `field` tactics of Coq to the commutative rings and fields of MathComp (Section 5.1). We demonstrated their practicality and scalability by applying them to the formal proof of Apéry's theorem (Section 5.3). Although their reflexive decision procedures are not our contribution, we found room for improvement on this point. For example, the `ring_exp` tactic [5] of the Lean [42] mathematical library [61] solves ring equations with variables in exponents, which is one of the cases we wished to solve in Section 5.3. The `ring_exp` tactic does not directly support homomorphisms, but the `simp` and `norm_cast` tactics [36] serve as preprocessors for pushing down and up homomorphisms as in Section 4.1 and 5.2. Since those Lean tactics can be performed alone as in the `ppsimpl` tactic, the above comparison of our approach and other preprocessing tools also applies here. Also, those Lean tactics are not reflexive and produce proof terms explaining rewriting steps. While such implementation does not require proving the procedure correct and is regarded as performance-wise better than reflection in Lean, it would not scale up to large equations and expressions that have large normal forms. On the other hand, implementing efficient tactics using computational reflection requires verified and efficient procedures involving computation-oriented data structures such as sparse Horner form [31]. Cohen and Rouhling [20] proposed a modular approach to define and reason about efficient decision procedures using CoqEAL refinement framework [19, 23], which is a suitable candidate method for extensively developing reflexive tactics for mathematical structures of MathComp.

References

1 Reynald Affeldt, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, and Kazuhiko Sakaguchi. Competing inheritance paths in dependent type theory: A case study in functional analysis. In *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2020. `doi:10.1007/978-3-030-51054-1_1`.

2 Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William E. Aitken. The semantics of reflected proof. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990*, pages 95–105. IEEE Computer Society, 1990. `doi:10.1109/LICS.1990.113737`.

3 Roger Apéry. Irrationalité de $\zeta(2)$ et $\zeta(3)$. *Astérisque*, 61, 1979. Société Mathématique de France.

4 Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. Hints in unification. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2009. `doi:10.1007/978-3-642-03359-9_8`.

5 Anne Baanen. A Lean tactic for normalising ring expressions with exponents (short paper). In *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 21–27. Springer, 2020. `doi:10.1007/978-3-030-51054-1_2`.

6 Frédéric Besson. Fast reflexive arithmetic tactics the linear case and beyond. In *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, volume 4502 of *Lecture Notes in Computer Science*, pages 48–62. Springer, 2006. `doi:10.1007/978-3-540-74464-1_4`.

7 Frédéric Besson. ppsimpl: a reflexive Coq tactic for canonising goals. In *The Third International Workshop on Coq for Programming Languages, CoqPL 2017*, 2017. URL: `https://popl17.sigplan.org/details/main/3/ppsimpl-a-reflexive-Coq-tactic-for-canonising-goals`.

8 Frédéric Besson. Fix #10779 (hnf normalisation of instance + reification of overloaded operators), 2019. Accessed: 2022-04-27. URL: `https://github.com/coq/coq/pull/10787`.

9 Valentin Blot, Denis Cousineau, Enzo Crance, Louise Dubois de Prisque, Chantal Keller, Assia Mahboubi, and Pierre Vial. Modular pre-processing for automated reasoning in dependent type theory, 2022. `doi:10.48550/ARXIV.2204.02643`.

10 Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. Full reduction at full throttle. In *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, volume 7086 of *Lecture Notes in Computer Science*, pages 362–377. Springer, 2011. `doi:10.1007/978-3-642-25379-9_26`.

11 Samuel Boutin. Using reflection to build efficient and certified decision procedures. In *Theoretical Aspects of Computer Software, Third International Symposium, TACS '97, Sendai, Japan, September 23-26, 1997, Proceedings*, volume 1281 of *Lecture Notes in Computer Science*, pages 515–529. Springer, 1997. `doi:10.1007/BFb0014565`.

12 Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda - a functional language with dependent types. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78. Springer, 2009. `doi:10.1007/978-3-642-03359-9_6`.

13 Thomas Braibant and Damien Pous. Tactics for reasoning modulo AC in Coq. In *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, volume 7086 of *Lecture Notes in Computer Science*, pages 167–182. Springer, 2011. `doi:10.1007/978-3-642-25379-9_14`.

14 Thomas Braibant and Damien Pous. Deciding Kleene algebras in Coq. *Logical Methods in Computer Science*, 8(1), 2012. `doi:10.2168/LMCS-8(1:16)2012`.

15 Adam Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant.* MIT Press, 2013. URL: `http://mitpress.mit.edu/books/certified-programming-dependent-types`.

16 Frédéric Chyzak, Assia Mahboubi, and Thomas Sibut-Pinote. Apery: A formal proof of the irrationality of zeta(3), the Apéry constant, 2020. Accessed: 2022-02-08. URL: `https://github.com/coq-community/apery`.

17 Frédéric Chyzak, Assia Mahboubi, Thomas Sibut-Pinote, and Enrico Tassi. A computer-algebra-based formal proof of the irrationality of $\zeta(3)$. In *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8558 of *Lecture Notes in Computer Science*, pages 160–176. Springer, 2014. `doi:10.1007/978-3-319-08970-6_11`.

18 Cyril Cohen. *Formalized algebraic numbers: construction and first-order theory. (Formalisation des nombres algébriques : construction et théorie du premier ordre).* PhD thesis, École Polytechnique, Palaiseau, France, 2012. URL: `https://tel.archives-ouvertes.fr/pastel-00780446`.

19 Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free! In *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*, volume 8307 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2013. `doi:10.1007/978-3-319-03545-1_10`.

20 Cyril Cohen and Damien Rouhling. A refinement-based approach to large scale reflection for algebra. In *JFLA 2017 - Vingt-huitième Journées Francophones des Langages Applicatifs*, 2017. URL: `https://hal.inria.fr/hal-01414881`.

21 Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. Hierarchy builder: Algebraic hierarchies made easy in Coq with Elpi (system description). In *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference)*, volume 167 of *LIPIcs*, pages 34:1–34:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. `doi:10.4230/LIPIcs.FSCD.2020.34`.

22 David Delahaye. A tactic language for the system Coq. In *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, November 11-12, 2000, Proceedings*, volume 1955 of *Lecture Notes in Computer Science*, pages 85–95. Springer, 2000. `doi:10.1007/3-540-44404-1_7`.

23 Maxime Dénès, Anders Mörtberg, and Vincent Siles. A refinement-based approach to computational algebra in Coq. In *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*, volume 7406 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2012. `doi:10.1007/978-3-642-32347-8_7`.

24 Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. ELPI: fast, embeddable, λProlog interpreter. In *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, volume 9450 of *Lecture Notes in Computer Science*, pages 460–468. Springer, 2015. `doi:10.1007/978-3-662-48899-7_32`.

25 François Garillot. *Generic Proof Tools and Finite Group Theory. (Outils génériques de preuve et théorie des groupes finis).* PhD thesis, École Polytechnique, Palaiseau, France, 2011. URL: `https://tel.archives-ouvertes.fr/pastel-00649586`.

26 François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging mathematical structures. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 327–342. Springer, 2009. `doi:10.1007/978-3-642-03359-9_23`.

27 Paolo G. Giarrusso. ring tactic breaks on mathcomp: it requires all ring operations are defined on *syntactically* equal types, not definitionally equal ones, 2020. Accessed: 2021-12-07. URL: `https://github.com/coq/coq/issues/11998`.

28   Georges Gonthier and Enrico Tassi. A language of patterns for subterm selection. In *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*, volume 7406 of *Lecture Notes in Computer Science*, pages 361–376. Springer, 2012. `doi:10.1007/978-3-642-32347-8_25`.

29   Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. How to make ad hoc proof automation less ad hoc. *Journal of Functional Programming*, 23(4):357–401, 2013. `doi:10.1017/S0956796813000051`.

30   Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, pages 235–246. ACM, 2002. `doi:10.1145/581478.581501`.

31   Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in Coq. In *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 2005. `doi:10.1007/11541868_7`.

32   Jason Gross, Andres Erbsen, and Adam Chlipala. Reification by parametricity - fast setup for proof by reflection, in two lines of Ltac. In *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10895 of *Lecture Notes in Computer Science*, pages 289–305. Springer, 2018. `doi:10.1007/978-3-319-94821-8_17`.

33   Benjamin S. Hvass. ring tactic for math-comp integers, 2019. Accessed: 2021-12-07. URL: `https://github.com/math-comp/math-comp/issues/401`.

34   Daniel W. H. James and Ralf Hinze. A reflection-based proof tactic for lattices in Coq. In *Proceedings of the Tenth Symposium on Trends in Functional Programming, TFP 2009, Komárno, Slovakia, June 2-4, 2009*, volume 10 of *Trends in Functional Programming*, pages 97–112. Intellect, 2009.

35   Jan-Oliver Kaiser, Beta Ziliani, Robbert Krebbers, Yann Régis-Gianas, and Derek Dreyer. Mtac2: typed tactics for backward reasoning in Coq. *Proceedings of the ACM on Programming Languages*, 2(ICFP):78:1–78:31, 2018. `doi:10.1145/3236773`.

36   Robert Y. Lewis and Paul-Nicolas Madelaine. Simplifying casts and coercions (extended abstract). In *Joint Proceedings of the 7th Workshop on Practical Aspects of Automated Reasoning (PAAR) and the 5th Satisfiability Checking and Symbolic Computation Workshop (SC-Square) Workshop, 2020 co-located with the 10th International Joint Conference on Automated Reasoning (IJCAR 2020), Paris, France, June-July, 2020 (Virtual)*, volume 2752 of *CEUR Workshop Proceedings*, pages 53–62, 2020. URL: `http://ceur-ws.org/Vol-2752/paper4.pdf`.

37   Assia Mahboubi and Thomas Sibut-Pinote. A formal proof of the irrationality of $\zeta(3)$. *Logical Methods in Computer Science*, 17(1), 2021. `doi:10.23638/LMCS-17(1:16)2021`.

38   Assia Mahboubi and Enrico Tassi. Canonical structures for the working Coq user. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2013. `doi:10.1007/978-3-642-39634-2_5`.

39   Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Zenodo, 2021. `doi:10.5281/zenodo.4457887`.

40   Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012. `doi:10.1017/CBO9781139021326`.

41   Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021. `doi:10.1007/978-3-030-79876-5_37`.

42 Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015. `doi:10.1007/978-3-319-21401-6_26`.

43 Pierre-Marie Pédrot. Ltac2: Tactical warfare. In *The Fifth International Workshop on Coq for Programming Languages, CoqPL 2019*, 2019. URL: `https://popl19.sigplan.org/details/CoqPL-2019/8/Ltac2-Tactical-Warfare`.

44 Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, pages 199–208. ACM, 1988. `doi:10.1145/53990.54010`.

45 Claudio Sacerdoti Coen and Enrico Tassi. ELPI - embeddable λProlog interpreter. Accessed: 2022-02-08. URL: `https://github.com/LPCIC/elpi`.

46 Amokrane Saïbi. Typing algorithm in type theory with inheritance. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, pages 292–301. ACM Press, 1997. `doi:10.1145/263699.263742`.

47 Amokrane Saïbi. *Outils Génériques de Modélisation et de Démonstration pour la Formalisation des Mathématiques en Théorie des Types. Application à la Théorie des Catégories. (Formalization of Mathematics in Type Theory. Generic tools of Modelisation and Demonstration. Application to Category Theory)*. PhD thesis, Pierre and Marie Curie University, Paris, France, 1999. URL: `https://tel.archives-ouvertes.fr/tel-00523810`.

48 Kazuhiko Sakaguchi. Validating mathematical structures. In *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 138–157. Springer, 2020. `doi:10.1007/978-3-030-51054-1_8`.

49 Kazuhiko Sakaguchi. Algebra tactics, 2021. Accessed: 2022-02-08. URL: `https://github.com/math-comp/algebra-tactics`.

50 Kazuhiko Sakaguchi. Mczify, 2021. Accessed: 2022-02-08. URL: `https://github.com/math-comp/mczify`.

51 Matthieu Sozeau and Nicolas Oury. First-class type classes. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2008. `doi:10.1007/978-3-540-71067-7_23`.

52 Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science*, 21(4):795–825, 2011. `doi:10.1017/S0960129511000119`.

53 Enrico Tassi. Coq-Elpi. Accessed: 2022-02-08. URL: `https://github.com/LPCIC/coq-elpi`.

54 The Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2022. the PDF version with numbered sections is available at `https://doi.org/10.5281/zenodo.1003420`. URL: `https://coq.inria.fr/distrib/V8.15.0/refman/`.

55 The Coq Development Team. Section 2.2.4 in [54]: "Syntax extensions and notation scopes", 2022. URL: `https://coq.inria.fr/distrib/V8.15.0/refman/user-extensions/syntax-extensions`.

56 The Coq Development Team. Section 2.2.6 in [54]: "Implicit coercions", 2022. URL: `https://coq.inria.fr/distrib/V8.15.0/refman/addendum/implicit-coercions`.

57 The Coq Development Team. Section 2.2.8 in [54]: "Canonical structures", 2022. URL: `https://coq.inria.fr/distrib/V8.15.0/refman/language/extensions/canonical`.

58 The Coq Development Team. Section 3.1.5 in [54]: "The ssreflect proof language", 2022. URL: `https://coq.inria.fr/distrib/V8.15.0/refman/proof-engine/ssreflect-proof-language`.

59 The Coq Development Team. Section 3.2.2 in [54]: "Micromega: solvers for arithmetic goals over ordered rings", 2022. URL: `https://coq.inria.fr/distrib/V8.15.0/refman/addendum/micromega`.

60 The Coq Development Team. Section 3.2.3 in [54]: "Ring and field: solvers for polynomial and rational equations", 2022. URL: `https://coq.inria.fr/distrib/V8.15.0/refman/addendum/ring`.

61 The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 367–381. ACM, 2020. `doi:10.1145/3372885.3373824`.

62 Alfred van der Poorten. A proof that Euler missed: Apéry's proof of the irrationality of $\zeta(3)$. *Math. Intelligencer*, 1(4):195–203, 1979. An informal report. `doi:10.1007/BF03028234`.

63 Beta Ziliani and Matthieu Sozeau. A comprehensible guide to a new unifier for CIC including universe polymorphism and overloading. *Journal of Functional Programming*, 27:e10, 2017. `doi:10.1017/S0956796817000028`.