Projet PiCoq

Deliverable D4

December 2011

During year 2011, Damien Pous and Thomas Braibant developed two libraries to assist in the mechanization of formal proofs in Coq. The first library provides algebraic tools for reasoning about binary relations and can be found at `http://sardes.inrialpes.fr/~braibant/atbr/`. The second library helps reasoning modulo associativity and commutativity. This library is available at `http://sardes.inrialpes.fr/~braibant/aac_tactics/`. We attach to this deliverable two papers describing these libraries: “Deciding Kleene Algebras in Coq” for the first one, and “Tactics for Reasoning modulo AC in Coq” for the second one.

DECIDING KLEENE ALGEBRAS IN COQ

THOMAS BRAIBANT AND DAMIEN POUS

CNRS, INRIA, LIG, UMR 5217, Université de Grenoble, France, Europe
e-mail address: {thomas.braibant,damien.pous}@inria.fr

Abstract. We present a reflexive tactic for deciding the equational theory of Kleene algebras in the Coq proof assistant. This tactic relies on a careful implementation of efficient finite automata algorithms, so that it solves casual equations instantaneously and properly scales to larger expressions. The decision procedure is proved correct and complete: correctness is established w.r.t. any model by formalising Kozen's initiality theorem; a counter-example is returned when the given equation does not hold. The correctness proof is challenging: it involves both a precise analysis of the underlying automata algorithms and a lot of algebraic reasoning. In particular, we have to formalise the theory of matrices over a Kleene algebra. We build on the recent addition of first-class typeclasses in Coq in order to work efficiently with the involved algebraic structures.

1. Introduction

1.1. **Motivations.** Proof *assistants* like Coq or Isabelle/HOL make it possible to leave technical or administrative details to the computer, by defining high-level tactics. For example, one can define tactics to solve decidable problems automatically (e.g., `omega` for Presburger arithmetic and `ring` for ring equalities). Here we present a tactic for solving equations and inequations in Kleene algebras. This tactic belongs to a larger project whose aim is to provide tools for working with binary relations in Coq. Indeed, Kleene algebras correspond to a non-trivial decidable fragment of binary relations. In the long term, we plan to use these tools to formalise results in rewriting theory, process algebras, and concurrency theory results. Binary relations play a central role in the corresponding semantics.

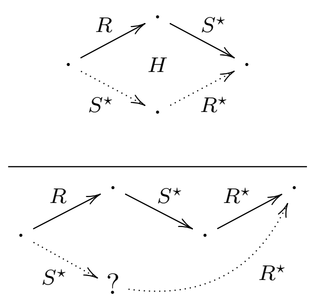
A starting point for this work is the following remark: proofs about abstract rewriting (e.g., Newman's Lemma, equivalence between weak confluence and the Church-Rosser property, termination theorems based on commutation properties) are best presented using informal "diagram chasing arguments". This is illustrated by Fig. 1, where the same state of a typical proof is represented three times. Informal diagrams are drawn on the left. The goal listed in the middle corresponds to a naive formalisation where the points related by

1998 ACM Subject Classification: F 1.1 [Finite automata], F 3.1 [Verification techniques] F.4.1 [Proof theory], F.4.3 [Decision problems], D 2.4 [Correctness proofs].

Key words and phrases: Kleene algebra, regular expressions, decision procedure, Coq proof assistant, reflexive tactic, finite automata, typeclasses.

This paper is a long version of the abstracts we presented at the first Coq workshop (August 2009) and in Proc. 1st ITP, vol. 6172 of LNCS, 2010 [10].

This work was supported by the project PiCoq (ANR 2010 BLAN 0305 01).



```
R,S: relation P
H: forall p,r,q, R p r → S* r q
    → exists s, S* p s ∧ R* s q
p,q,q', s: P
Hpq: R p q
Hqs: S* q s
Hsq': R* s q'
________________________________
exists s, S* p s ∧ R* s q'
```

```
R,S: X
H: R · S* ≤ S* · R*
___________________
R · S* · R* ≤ S* · R*
```

Figure 1. Diagrammatic, concrete, and abstract presentations of the same state in a proof.

relations are mentioned explicitly. This is not satisfactory: a lot of variables have to be introduced, the goal is displayed in a rather verbose way, the user has to draw the intuitive diagrams on its own paper sheet. On the contrary, if we move to an algebraic setting (the right-hand side goal), where binary relations are seen as abstract objects that can be composed using various operators (e.g., union, intersection, relational composition, iteration), statements and Coq's output become rather compact, making the current goal easier to read and to reason about.

More importantly, moving to such an abstract setting allows us to implement several decision procedures that could hardly be stated with the concrete presentation. For example, after the user rewrites the hypothesis `H` in the right-hand side goal of Fig. 1, we obtain the inclusion `S* · R* · R*≤ S* · R*`, which is a (straightforward) theorem of Kleene algebras: the tactic we describe in this paper proves this sub-goal automatically.

1.2. **Mathematical background.** A Kleene algebra [38] is a tuple $\langle X, \cdot, +, 1, 0, \star\rangle$, where $\langle X, \cdot, +, 1, 0\rangle$ is an idempotent non-commutative semiring, and $\_^\star$ is a unary post-fix operation on X, satisfying the following axiom and inference rules (where $\leq$ is the partial order defined by $x \leq y \;\triangleq\; x + y = y$):

$$1 + x \cdot x^\star = x^\star \qquad\qquad \frac{x \cdot y \leq y}{x^\star \cdot y \leq y} \qquad\qquad \frac{y \cdot x \leq y}{y \cdot x^\star \leq y}$$

Terms of Kleene algebras, ranged over using x, y, are called *regular expressions*, irrespective of the considered model. Models of Kleene algebras include *languages*, where the unit (1) is the language reduced to the empty word, product $(\cdot)$ is language concatenation, and star $(\_^\star)$ is language iteration; and *binary relations*, where the unit is the identity relation, product is relational composition, and star is reflexive and transitive closure. Here are some theorems of Kleene algebras:

$$x^\star = x^\star \cdot x^\star = x^{\star\star} = (x+1)^\star \qquad (x+y)^\star = x^\star \cdot (y \cdot x^\star)^\star \qquad x \cdot (y \cdot x)^\star = (x \cdot y)^\star \cdot x$$

Among languages, those that can be described by a finite state automaton (or equivalently, generated by a regular expression) are called *regular*. Thanks to finite automata theory [37, 49], equality of regular languages is decidable:

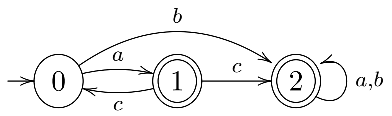
> *"two regular languages are equal if and only if the corresponding minimal automata are isomorphic".*

However, the above theorem is not sufficient to derive equations in all Kleene algebras: it only applies to the model of regular languages. We actually need a more recent theorem, by Kozen [38] (independently proved by Krob [43]):

> *"if two regular expressions x and y denote the same regular language, then $x = y$ is a theorem of Kleene algebras".*

In other words, the algebra of regular languages is initial among Kleene algebras: we can use finite automata algorithms to solve equations in an arbitrary Kleene algebra.

The main idea of Kozen's proof is to encode finite automata using matrices over regular expressions, and to replay the algorithms at this algebraic level. Indeed, a finite automaton can be represented with three matrices $\langle u, M, v\rangle \in \mathcal{M}_{1,n} \times \mathcal{M}_{n,n} \times \mathcal{M}_{n,1}$: n is the number of states of the automaton, u and v are 0-1 vectors respectively coding for the sets of initial and accepting states, and M is the transition matrix: $M_{i,j}$ labels transitions from state i to state j. Consider for example the following non-deterministic automaton, with three states (like for the automata to be depicted in the sequel, accepting states are marked with two circles, and short, unlabelled arrows point to the starting states):



This automaton can be represented using the following matrices:

$$u = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \qquad M = \begin{bmatrix} 0 & a & b \\ c & 0 & c \\ 0 & 0 & a+b \end{bmatrix} \qquad v = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} .$$

We can remark that the product $u \cdot M \cdot v$ is a scalar (i.e., a regular expression), which can be thought of as the set of one-letter words accepted by the automaton—in the example, $a+b$. Similarly, $u \cdot M^2 \cdot v = u \cdot M \cdot M \cdot v$ corresponds to the set of two-letters words accepted by the automaton—here, $a \cdot c + b \cdot a + b \cdot b$. Therefore, to mimic the behaviour of a finite automaton and get the whole language it accepts, we just need to iterate over the matrix M. This is possible thanks to another theorem, which actually is the crux of the initiality theorem: *"matrices over a Kleene algebra form a Kleene algebra"*. We hence have a star operation on matrices, and we can interpret an automaton algebraically, by considering the product $u \cdot M^\star \cdot v$. (Again, in the example, we could check that this computation reduces into a regular expression which is equivalent to $(a \cdot c)^\star \cdot (a + (b + a \cdot c) \cdot (a + b)^\star)$, which corresponds precisely to the language accepted by the automaton.)

1.3. **Overview of our strategy.**

We define a *reflexive* tactic. This methodology is quite standard [8, 2]. For example, this is how the `ring` tactic is implemented [29]. Concretely, this means that we program the decision procedure as a Coq function, and that we prove its correctness and its completeness within the proof assistant:

```
Definition decide_kleene: regex → regex → bool := ...
Theorem Kozen94: forall x y: regex, decide_kleene x y = true ↔ x == y.
```

The above statement corresponds to correctness and completeness with respect to the syntactic "free" Kleene algebra: `regex` is the inductive type for regular expressions over a countable set of variables, and `==` is the inductive equality generated by the axioms of

Kleene algebras and the rules of equational reasoning. Using reification mechanisms, this is sufficient for our needs: the result can be lifted to other models using simple tactics.

Here are the main requirements we had to take into account for the design of the library:

Efficiency. The equational theory of Kleene algebras is PSPACE-complete [46]; this means that the `decide_kleene` function must be written with care, using efficient algorithms. Notably, the matricial representation of automata is not efficient, so that formalising Kozen's "mathematical" proof [38] in a naive way would be computationally impracticable. Instead, we need to choose appropriate data structures for automata and algorithms, and to rely on the matricial representation only in proofs, using the adequate translation functions.

Heterogeneous models. Homogeneous binary relations are a model of Kleene algebras, but binary relations can be heterogeneous: their domain might differ from their co-domain so that they fall out of the scope of standard Kleene algebra. We could use a trick to handle the special case of heterogeneous relations [42], but there is a more general and more algebraic solution that captures all heterogeneous models: it suffices to consider the rather natural notion of *typed* Kleene algebra [39]. Since we want to put forward the algebraic approach, we tend to prefer this second option. Moreover, as pointed out in next paragraph, we actually exploit this generalisation to formalise Kozen's proof.

Matrices. As explained in Sect. 1.2, Kozen's proof relies on the theory of matrices over regular expressions, which we thus need to formalise. First, this formalisation must be tractable from the proof point of view: the overall proof requires a lot of matricial reasoning. Second, we must handle rectangular matrices, which appear in some parts of the proof (see Sect. 4.4). The latter point can be achieved in a nice way thanks to the generalisation to typed Kleene algebra: while only square matrices of form a model of Kleene algebra, rectangular matrices form a model of typed Kleene algebra.

Sharing. The overall proof being rather involved, we need to exploit sharing as much as possible. For instance, we work with several models of Kleene algebra (the syntactic model of regular expressions, matrices over regular expressions, languages, matrices over languages, and relations). Since these models share the same properties, we need to share notation, basic laws, theorems, and tactics: this improves readability, usability, and maintainability. Similarly, the proof requires vectors, which we define as a special case of (rectangular) matrices: this saves us from re-developing their theory separately.

Modularity. Following mathematical and programming practice, we aim at a modular development: this is required to be able to get sharing between the various parts of the proof. A typical example is the definition of the Kleene algebra of matrices (Sect. 3.3), which corresponds to a rather long proof. With a monolithic definition of Kleene algebra, we would have to prove that all axioms of Kleene algebra hold *from scratch*. On the contrary, with a modular definition, we can first prove that matrices form an idempotent semiring, which allows us to use theorems and tactics about semiring when proving that the defined star operation actually satisfies the appropriate laws.

Reification. The final tactic (for deciding Kleene algebras) and some intermediate tactics are defined by reflection. Therefore, we need a way to achieve *reification*, i.e., to transform a goal into a reified version that lets us perform computations within Coq. Since we work with *typed* models, this step is more involved than is usually the case.

Outline of the paper. Section 2 is devoted to the underlying design choices. We explain how we define matrices in Sect. 3. The algorithm and its correctness proof are described in Sect. 4. We discuss the efficiency of the tactic in Sect. 5. We conclude with related works and directions for future work in Sect. 6.

2. Underlying design choices

According to the above constraints and objectives, an essential decision was to build on the recent introduction of first-class *typeclasses* in Coq [52]. This section is devoted to the explanation of our methodology: how to use typeclasses to define the algebraic hierarchy in a modular way, how to formalise typed algebras, how to reify the corresponding expressions. We start with a brief description of the implementation of typeclasses in Coq.

2.1. Basic introduction to typeclasses in Coq.

The overall behaviour of Coq typeclasses [52] is quite intuitive; here is how we would translate to Coq a simple Haskell program that exploits a typeclass `Hash` to get a number out of certain kind of values:

```
class Hash a where
 hash :: a → Int
instance Hash Int where
 hash = id
instance (Hash a) ⇒ (Hash [a]) where
 hash = sum . map hash
main = print
 (hash 4, hash [4,5,6],  hash [[4,5],[]])
```

```
Class Hash A :=
 { hash: A → nat }.
Instance hash_n: Hash nat :=
 { hash x := x }.
Instance hash_l A: Hash A → Hash (list A) :=
 { hash l := fold_left (fun a x ⇒ a + hash x) l 0 }.
Eval simpl in
 (hash 4, hash [4;5;6],  hash [[4;5];[]]).
```

Coq typeclasses are first-class; everything is done with plain Coq terms. In particular, the `Class` keyword produces a record type (here, a parametrised one) and the `Instance` keyword acts like a standard definition. With the above code we get values of the following types:

```
Hash: Type → Type                     hash_n: Hash nat
hash: forall A, Hash A → A → nat      hash_l: forall A, Hash A → Hash (list A)
```

The function `hash` is a *class projection*: it gives access to a field of the class. The subtlety is that the first two arguments of this function are implicit: they are automatically inserted by unification and typeclass resolution. More precisely, when we write "`hash` [4;5;6]", Coq actually reads "@`hash _ _` [4;5;6]" (the '@name' syntax can be used in Coq to give all arguments explicitly). By unification, the first placeholder has to be `list nat`, and Coq needs to guess a term of type `Hash (list nat)` to fill the second placeholder. This term is obtained by a simple proof search, using the two available instances for the class `Hash`, which yields "@`hash_l nat hash_n`". Accordingly, we get the following explicit terms for the three calls to `hash` in the above example.

input term	explicit, instantiated, term
`hash` 4	@`hash nat hash_n` 4
`hash` [4;5;6]	@`hash (list nat)` (@`hash_l nat hash_n`) [4;5;6]
`hash` [[4;5];[]]	@`hash (list (list nat))` (@`hash_l (list nat) (hash_l nat hash_n)`) [[4;5];[]]

```
X: Type.

dot:  X → X → X.
one:  X.
plus: X → X → X.
zero: X.
star: X → X.

dot_neutral_left:
 forall x, dot one x = x.
...
```

```
T: Type.
X: T → T → Type.

dot:  forall n m p, X n m → X m p → X n p.
one:  forall n,     X n n.
plus: forall n m,   X n m → X n m → X n m.
zero: forall n m,   X n m.
star: forall n,     X n n → X n n.

dot_neutral_left:
 forall n m (x: X n m),  dot one x = x.
...
```

Figure 2. From Kleene algebras to typed Kleene algebras.

In summary, typeclasses provide overloading (we can use the `hash` function on several types) and allow one to write much shorter and readable terms, by letting Coq infer the obvious boilerplate. This concludes our very short introduction to typeclasses in Coq; we invite the reader to consult [52] for more details.

2.2. **Using typeclasses to structure the development.** We use typeclasses to achieve two tasks: 1) sharing and overloading notation, basic laws, and theorems; 2) getting a modular definition of Kleene algebra, by mimicking the standard mathematical hierarchy: a Kleene algebra contains an idempotent semiring, which is itself composed of a monoid and a semi-lattice. This very small hierarchy is summarised below.

```
SemiLattice  <:
   Monoid    <:  IdemSemiRing <: KleeneAlgebra
```

Before we give concrete Coq definitions, recall that we actually want to work with the *typed* versions of the above algebraic structures, to be able to handle both heterogeneous binary relations and rectangular matrices. The intuition for moving from untyped structures to typed structures is given in Fig. 2: a typical signature for Kleene algebras is presented on the left-hand side; we need to move to the signature on the right-hand side, where a set `T` of indices (or types) is used to restrain the domain of the various operations. These indices can be thought of as matrix dimensions; we actually moved to a categorical setting: `T` is a set of objects, `X n m` is the set of morphisms from `n` to `m`, `one` is the set of identities, and `dot` is composition. The semi-lattice operations (`plus` and `zero`) operate on fixed homsets; Kleene star operates only on *square* morphisms—those whose source and target coincide.

Classes for algebraic operations. We now can define the Coq classes on which we based our library. We first define three classes, for the operations corresponding to a monoid, a semi-lattice, and Kleene star. These classes are given in Fig. 3, they are parametrised by a fourth class, `Graph`, which corresponds to the carrier of the algebraic operations. In a standard, untyped setting, we would expect this carrier to be just a set (a `Type`); the situation is slightly more complicated here, since we define typed algebraic structures. According to the previous explanations and Fig. 2, the `Graph` class encapsulates several ingredients: a

```
Class Graph := {
 T: Type;
 X: T → T → Type;
 equal: ∀ n m, relation (X n m);
 equal_:> ∀ n m, Equivalence (equal n m) }.

Class Monoid_Ops (G: Graph) := {
 dot: ∀ n m p, X n m → X m p → X n p;
 one: ∀ n,      X n n }.

Notation "x == y" := (equal _ _ x y).
Notation "x  ·  y" := (dot _ _ _ x y).
Notation "1"       := (one _).
```

```
Class SemiLattice_Ops (G: Graph) := {
 plus: ∀ n m, X n m → X n m → X n m;
 zero: ∀ n m, X n m }.

Class Star_Op (G: Graph) := {
 star: ∀ n, X n n → X n n }.

Notation "x  + y" := (plus _ _ x y).
Notation "0"      := (zero _ _).
Notation "x⋆"     := (star _ x).
Notation "x ≤ y" := (x  + y == y).
```

Figure 3. Classes for the typed algebraic operations.

type for the set of indices (`T`), an indexed family of types for the sets of morphisms (`X`), and for each homset, an equivalence relation, `equal`—we cannot use Leibniz equality: most models of Kleene algebra require a weaker notion of equality (`relation` and `Equivalence` are definitions from the standard library).

We associate an intuitive notation to each operation, by using the name provided by the corresponding class projection. To make the effect of these definitions completely clear, assume that we have a graph equipped with monoid operations (i.e., a typing context with `G: Graph` and `Mo: Monoid_Ops G`) and consider the following proposition:

```
∀ (n m: T) (x: X n m) (y: X m n), x · y == 1.
```

If we unfold notations, we get:

```
∀ (n m: T) (x: X n m) (y: X m n), equal _ _ (dot _ _ _ x y) (one _).
```

Necessarily, by unification, the six placeholders have to be filled as follows:

```
∀ (n m: T) (x: X n m) (y: X m n), equal n n (dot n m n x y) (one n).
```

Now comes typeclass resolution: as explained in Sect. 2.1, the functions `T`, `X`, `equal`, `dot`, and `one`, which are class projections, have implicit arguments that are automatically filled by typeclass resolution (the graph instance for all of them, and the monoid operations instance for `dot` and `one`). All in all, the above concise proposition actually expands into:

```
∀ (n m: @T G) (x: @X G n m) (y: @X G m n), @equal G n n (@dot G Mo n m n x y) (@one G Mo n).
```

Classes for algebraic laws. This was for syntax; we can finally define the classes for the laws corresponding to the four algebraic structures we are interested in. They are given in Fig. 4; we use the section mechanism to assume a graph together with the operations, which become parameters when we close the section. (We motivate our choice to have separate classes for operations and for laws in Sect. 2.4.3.)

The `Monoid` class actually corresponds to the definition of a category: we assume that composition (`dot`) is associative and has `one` as neutral element. Its first field, `dot_compat`, requires that composition also preserves the user-defined equality: it has to map equals

```
Section.
 Context (G: Graph) {Mo: Monoid_Ops G} {SLo: SemiLattice_Ops G} {Ko: Star_Op G}.

 Class Monoid := {
   dot_compat:> ∀ n m p, Proper (equal n m ==> equal m p ==> equal n p) (dot n m p);
   dot_assoc: ∀ n m p q (x: X n m) (y: X m p) (z: X p q), x · (y · z) == (x · y) · z;
   dot_neutral_left: ∀ n m (x: X n m), 1 · x == x;
   dot_neutral_right: ∀ n m (x: X m n), x · 1 == x }.

 Class SemiLattice := {
   plus_compat:> ∀ n m, Proper (equal n m ==> equal n m ==> equal n m) (plus n m);
   plus_neutral_left: ∀ n m (x: X n m), 0 + x == x;
   plus_idem: ∀ n m (x: X n m), x + x == x;
   plus_assoc: ∀ n m (x y z: X n m), x + (y + z) == (x + y) + z;
   plus_com: ∀ n m (x y: X n m), x + y == y + x }.

 Class IdemSemiRing := {
   Monoid_:> Monoid;
   SemiLattice_:> SemiLattice;
   dot_ann_left: ∀ n m p (x: X m p), 0 · x == (0: X m n);
   dot_ann_right: ∀ n m p (x: X p m), x · 0 == (0: X n m);
   dot_distr_left: ∀ n m p (x y: X n m) (z: X m p), (x + y) · z == x · z  + y · z;
   dot_distr_right: ∀ n m p (x y: X m n) (z: X p m), z · (x + y) == z · x  + z · y }.

 Class KleeneAlgebra := {
   IdemSemiRing_:> IdemSemiRing;
   star_make_left: ∀ n (x: X n n), 1  + x⋆ · x == x⋆;
   star_destruct_left: ∀ n m (x: X n n) (y: X n m), x · y ≤ y → x⋆ · y ≤ y;
   star_destruct_right: ∀ n m (x: X n n) (y: X m n), y · x ≤ y → y · x⋆ ≤ y }.
End.
```

Figure 4. Classes for the typed algebraic structures.

to equals. (This field is declared with a special symbol (:>) and uses the standard `Proper` class, which is exploited by Coq to perform rewriting with user-defined relations; doing so adds `dot_compat` as a hint for typeclass resolution, so that we can automatically rewrite in `dot` operands whenever it makes sense.) Also note that since this class does not mention semi-lattice operations nor the star operation, it does not depend on `SLo` and `Ko` when we close the section. We do not comment on the `SemiLattice` class, which is quite similar.

The first two fields of `IdemSemiRing` implement the expected inheritance relationship: an idempotent semiring is composed of a monoid and a semi-lattice whose operations properly distribute. By declaring these two fields with a :>, the corresponding projections are added as hints to typeclass resolution, so that one can automatically use any theorem about monoids or semi-lattices in the context of a semiring. Note that we have to use type annotations for the two annihilation laws: in both cases, the argument `n` of 0 (`zero`) cannot be inferred from the context, it has to be specified.

Finally, we obtain the class for Kleene algebras by inheriting from `IdemSemiRing` and requiring the three laws about Kleene star to hold. The counterpart of `star_make_left` and the fact that Kleene star is a proper morphism for `equal` are consequences of the other axioms; this is why we do not include a `star_compat` or `star_make_right` field in the signature: we prove these lemmas separately (and we declare the former as an instance for typeclass resolution), this saves us from additional proofs when defining new models.

The following example illustrates the ease of use of this approach. Here is how we would state and prove a lemma about idempotent semirings:

```
Goal forall `{IdemSemiRing} n (x y: X n n), x · (y + 1) + x == x · y + x.
Proof.
  intros.
  rewrite dot_distr_right, dot_neutral_right. (** (x · y + x) + x == x · y + x **)
  rewrite ← plus_assoc, plus_idem.
  reflexivity.
Qed.
```

The special `{IdemSemiRing} notation allows us to assume a generic idempotent semiring, with all its parameters (a graph, monoid operations, and semi-lattice operations); when we use lemmas like `dot_distr_right` or `plus_assoc`, typeclass resolution automatically finds appropriate instances to fill their implicit arguments. Of course, since such simple and boring goals occur frequently in larger and more interesting proofs, we actually defined high-level tactics to solve them automatically. For example, we have a reflexive tactic called `semiring_reflexivity` which would solve this goal directly: this is the counterpart to `ring` [29] for the equational theory of typed, idempotent, non-commutative semirings.

Declaring new models. It remains to populate the above classes with concrete structures, i.e., to declare models of Kleene algebra. We sketched the case of heterogeneous binary relations and languages in Fig. 5; a user needing its own model of Kleene algebra just has to declare it in the very same way. As expected, it suffices to define a graph equipped with the various operations, and to prove that they validate all the axioms. The situation is slightly peculiar for languages, which form an *untyped* model: although the instances are parametrised by a set `A` coding for the alphabet, there is no notion of domain/co-domain of a language. In fact, all operations are total, they actually lie in a one-object category where domain and co-domain are trivial. Accordingly, we use the singleton type `unit` for the index type `T` in the graph instance, and all operations just ignore the superfluous parameters.

2.3. **Reification: handling typed models.** We also need to define a syntactic model in which to perform computations: since we define a reflexive tactic, the first step is to reify the goal (an equality between two expressions in an arbitrary model) to use a syntactical representation.

For instance, suppose that we have a goal of the form $\mathtt{S} \cdot (\mathtt{R} \cdot \mathtt{S})^\star + \mathtt{f\ R} == \mathtt{f\ R} + (\mathtt{S} \cdot \mathtt{R})^\star \cdot \mathtt{S}$, where `R` and `S` are binary relations and `f` is an arbitrary function on relations. The usual methodology in Coq consists in defining a syntax and an evaluation function such that this goal can be converted into the following one:

$$\mathtt{eval}\ (\mathtt{var}\ 1 \odot (\mathtt{var}\ 2 \odot \mathtt{var}\ 1)^{\circledast} \oplus \mathtt{var}\ 3) == \mathtt{eval}\ (\mathtt{var}\ 3\ \oplus (\mathtt{var}\ 1 \odot \mathtt{var}\ 2)^{\circledast} \odot \mathtt{var}\ 1),$$

```
Definition rel A B := A → B → Prop.
Instance rel_G: Graph := {
  T := Type;
  X := rel;
  equal A B R S := ∀ i j, R i j ↔ S i j }.
Proof...
Instance rel_Mo: Monoid_Ops rel_G := {
  dot A B C R S :=
    fun (i: A)(j: C) ⇒ ∃ k: B, R i k ∧ S k j;
  one A :=
    fun (i j: A) ⇒ i=j }.
...
Instance rel_KA: KleeneAlgebra rel_G.
Proof...
```

```
Definition lang A := list A → Prop
Instance lang_G A: Graph := {
  T := unit;
  X _ _ := lang A;
  equal _ _ L K := ∀ w, L w ↔ K w }.
Proof...
Instance lang_Mo A: Monoid_Ops (lang_G A) := {
  dot _ _ _ L K :=
    fun w ⇒ ∃ u v, w=u++v ∧ L u ∧ K v;
  one _ :=
    fun w ⇒ w=[] }.
...
Instance lang_KA: KleeneAlgebra lang_G.
Proof...
```

Figure 5. Instances for heterogeneous binary relations and languages.

```
Context `{KA: KleeneAlgebra}.
Variables src, tgt: label → T.
Inductive reified: T → T → Type :=
| r_dot: ∀ n m p,
    reified n m → reified m p → reified n p
| r_one: ∀ n, reified n n
| ...
| r_var: ∀ i, reified (src i) (tgt i).
```

```
Variable env: forall i, X (src i) (tgt i).
Fixpoint eval n m (x: reified n m): X n m :=
  match x with
  | r_dot _ _ _ x y ⇒ eval x · eval y
  | r_one _ ⇒ 1
  | ...
  | r_var i ⇒ env i
  end.
```

Figure 6. Typed syntax for reification and evaluation function.

where $\oplus$, $\odot$, and $^{\circledast}$ are syntactic constructors, and where `eval` implicitly uses a reification environment, which corresponds to the following assignment:

$$\{1 \mapsto \texttt{S};\ 2 \mapsto \texttt{R};\ 3 \mapsto \texttt{f R}\}.$$

Typed syntax. The situation is slightly more involved here since we work with typed models: `R` might be a relation from a set `A` to another set `B`, `S` and `f R` being relations from `B` to `A`. As a consequence, we have to keep track of domain/co-domain information when we define the syntax and the reification environments. The corresponding definitions are given in Fig. 6. We assume an arbitrary Kleene algebra (in the previous example, it would be the algebra of heterogeneous binary relations) and two functions `src` and `tgt` associating a domain and a co-domain to each variable (`label` is an alias for `positive`, the type of positive numbers, which we use to index variables). The `reified` inductive type corresponds to the typed reification syntax: it has dependently typed constructors for all operations of Kleene algebras, and an additional constructor for variables, which is typed according to functions `src` and `tgt`. To define the evaluation function, we furthermore assume an assignation `env` from variables to elements of the Kleene algebra with domain and co-domain as specified by `src` and `tgt`. Reifying a goal using this typed syntax is relatively easy: thanks to

```
Inductive regex: Set :=
| dot:  regex → regex → regex
| plus: regex → regex → regex
| star: regex → regex
| one:  regex
| zero: regex
| var:  label → regex.

Inductive eq: regex → regex → Prop :=
| eq_trans: forall y x z, x== y → y== z → x== z
| plus_idem: forall x, eq (x + x) x
| plus_compat: Proper (eq ==> eq ==> eq) plus
| star_make_left: forall x, eq (1 + x⋆ · x) (x⋆)
| ...
```

```
Instance re_G: Graph := {
 T := unit;
 X _ _ := regex;
 equal _ _ := eq }.
Proof...

Instance re_Mo: Monoid_Ops re_G := {
 dot _ _ _ := dot;
 one _ := one }.

...

Instance re_KA: KleeneAlgebra re_G.
Proof...
```

Figure 7. Regular expressions, axiomatic equality, and corresponding instances.

the typeclass framework, it suffices to parse the goal, looking for typeclass projections to detect operations of interest (recall for example that a starred sub-term is always of the form `@star _ _ _ _`, regardless of the current model—this model is given in the first two placeholders). At first, we implemented this step as a simple `Ltac` tactic. For efficiency reasons, we finally moved to an OCaml implementation in a small plugin: this allows one to use efficient data structures like hash-tables to compute the reification environment, and to avoid type-checking the reified terms at each step of their construction.

Untyped regular expressions. To build a reflexive tactic using the above syntax, we need a theorem of the following form (keeping the reification environment implicit for the sake of readability):

```
Theorem f_correct: forall n m (x y: reified n m), f x y = true → eval x == eval y.
```

The function `f` is the decision procedure; it works on reified terms so that its type has to be `forall n m, reified n m → reified n m → bool`. However, defining such a function directly would be rather impractical: the standard algorithms underlying the decision procedure are essentially untyped, and since these algorithms are rather involved, extending them to take typed regular expressions into account would require a lot of work.

Instead, we work with standard, untyped, regular expressions, as defined by the inductive type `regex` from Fig. 7. Equality of regular expressions is defined inductively, using the rules from equational logic and the laws of Kleene algebra. By declaring the corresponding instances, we get an untyped model (on the right-hand side of Fig. 8—like for languages, we just ignore domain/co-domain information). This is the main model we shall work with to implement the decision procedure and prove its correctness (Sect. 4): as announced in Sect. 1.3, we will get:

```
Definition decide_kleene: regex → regex → bool := ...
Theorem Kozen94: forall x y: regex, decide_kleene x y = true ↔ x == y.
```

(Here the symbol `==` expands to the inductive equality predicate `eq` from Fig. 7.)

```
Fixpoint erase n m (x: reified n m): regex :=
  match x with
  | r_dot _ _ _ x y ⇒ erase x · erase y
  | r_one _ ⇒ 1
  | ...
  | r_var i ⇒ var i
  end.
```

```
Theorem erase_faithful:
  forall n m (x y: reified n m),
    erase x == erase y → eval x == eval y.
Proof...
```

Figure 8. Type erasing function and untyping theorem.

Untyping theorem. We still have to bridge the gap between this untyped decision procedure (to be presented in Sect. 4) and the reification process we described for typed models. To this end, we exploit a nice property of the equational theory of typed Kleene algebra: it reduces to the equational theory of untyped Kleene algebra [48]. In other words, a typed law holds in all typed Kleene algebras if and only if the underlying untyped law holds in all Kleene algebras.

To state this result formally, it suffices to define the type-erasing function `erase` from Fig. 8: this function recursively removes all type decorations of a typed regular expression to get a plain regular expression. The corresponding "untyping theorem" is given on the right-hand side: two typed expressions whose images under `erase` are equal in the model of untyped regular expressions evaluate to equal values in any typed model, under any variable assignation (again, the reification environment is left implicit here). By composing this theorem with the correctness of the untyped decision procedure—the previous theorem `Kozen94`, we get the following corollary, which allows us to get a reflexive tactic for typed models even though the decision procedure is untyped.

```
Corollary dk_erase_correct: forall n m (x y: reified n m),
  decide_kleene (erase x) (erase y) = true → eval x == eval y.
```

Proving the untyping theorem is non-trivial, it requires the definition of a proof factorisation system; see [48] for a detailed proof and a theoretical study of other untyping theorems. Also note that Kozen investigated a similar problem [39] and came up with a slightly different solution: he solves the case of the Horn theory rather than the equational theory, at the cost of working in a restrained form of Kleene algebras. He moreover relies on model-theoretic arguments, while our considerations are purely proof-theoretic.

Finally note that as it is stated here, theorem `erase_faithful` requires the axiom `Eqdep.eq_rect_eq` from Coq standard library. This comes from the inductive type `reified` from Fig. 6, which has dependent parameters in an arbitrary type (more precisely, the field `T` of an arbitrary graph `G`). We get rid of this axiom in the library at the price of an indirection: we actually make this inductive type depend on positive numbers and we use an additional map to enumerate the elements of `T` that are actually used (since terms are finite, there are only finitely many such elements in a given goal). Since the type of positive numbers has decidable equality, we can eventually avoid using axiom `Eqdep.eq_rect_eq` [30].

2.4. More details on our approach.

We conclude this section with additional remarks on the advantages and drawbacks of our design choices; the reader may safely skip these and move directly to Sect. 3.

```
Context {G: Graph} {Mo: Monoid_Ops G}
        {SLo: SemiLattice_Ops G}
        {Ko:  Star_Op G}.

Instance G': Graph := {
 T := T;
 X n m := X m n;
 equal n m := equal m n;
 equal_ n m := equal_ m n }.

Instance Mo': Monoid_Ops G' := {
 dot n m p x y := @dot G Mo p m n y x;
 one := @one G Mo }.

Instance SLo': SemiLattice_Ops G' := {
 plus n m := @plus G SLo m n;
 zero n m := @zero G SLo m n }.

Instance Ko': Star_Op G' := {
 star := @star G Ko }.
```

```
Instance M' {M: Monoid G}: Monoid G' := {
 dot_neutral_left n m :=
   @dot_neutral_right G Mo M m n;
 dot_neutral_right n m :=
   @dot_neutral_left G Mo M m n;
 dot_compat n m p x x' Hx y y' Hy :=
   @dot_compat G Mo M p m n y y' Hy x x' Hx }.
Proof.
 intros. symmetry. simpl. apply dot_assoc.
Qed.

...

Instance KA' {KA: KleeneAlgebra G}:
                          KleeneAlgebra G' := {
 star_destruct_left n m :=
   @star_destruct_right G Mo SLo Ko KA m n;
 star_destruct_right n m :=
   @star_destruct_left G Mo SLo KA m n }.
Proof...
```

Figure 9. Instances for the dual Kleene algebra.

2.4.1. *Taking advantage of symmetry arguments.* It is common practice in mathematics to rely on symmetry arguments to avoid repeating the same proofs again and again. Surprisingly, by carefully designing our classes and defining appropriate instances, we can also take advantage of some symmetries present in Kleene algebra, in a formal and simple way.

The starting point is the following observation. Consider a typed Kleene algebra as a category with additional structure on the homsets; by formally reversing all arrows, we get a new typed Kleene algebra. Therefore, any statement that holds in all typed Kleene algebra can be reversed, yielding another universally true statement. (This *duality principle* is standard in category theory [45]; it is also used in lattice theory [21], where we can always consider the dual lattice.)

In Coq, it suffices to define instances corresponding to this dual construction. These instances are given in Fig. 9. The dual graph and operations are obtained by swapping domains with co-domains; we get composition by furthermore reversing the order of the arguments. Proving that these reversed operations satisfy the laws of a Kleene algebra is relatively easy since almost all laws already come with their dual counterpart (we actually wrote laws with some care to ensure that the dual operation precisely maps such laws to their counterpart). The two exceptions are associativity of composition, which is in a sense self-dual up to symmetry of equality, and `star_make_left` whose dual is a consequence of the other axioms, so that it was not included in the signature of Kleene algebras— Fig. 4. (Note that these instances are dangerous from the typeclass resolution point of view: they introduce infinite paths in the proof search trees. Therefore, we do not export them and we use them only on a case by case basis.)

With these instances defined, suppose that we have proved

```
Lemma iter_right `{KA: KleeneAlgebra}: ∀ n m x y (z: X n m), z · x ≤ y · z → z · x⋆ ≤ y⋆ · z.
```

By symmetry we immediately get

```
Lemma iter_left `{KA: KleeneAlgebra}: ∀ n m x y (z: X m n), x · z ≤ z · y → x⋆ · z ≤ z · y⋆.
Proof iter_right (KA:=KA').
```

Indeed, instantiating the Kleene algebra with its dual in lemma `iter_right` amounts to swapping domains and co-domains in the type of variables (only `z` is altered since `x` and `y` have square types) and reversing the order of all products. Doing so, we precisely get the statement of lemma `iter_left`, up to conversion.

By combining the above two lemmas, we finally get the following one, which we actually use in Sect. 4.4.

```
Lemma iter `{KA: KleeneAlgebra}: ∀ n m x y (z: X n m), x · z == z · y → x⋆ · z == z · y⋆.
Proof...
```

2.4.2. *Concrete structures.* Our typeclass-based approach may become problematic when dealing with concrete structures without using our notations in a systematic way. This might be a drawback for potential end-users of the library. Indeed, suppose one wants to use a concrete type rather than our uninformative projection `X` to quantify over some relation `R` between natural numbers:

```
Check forall R: rel nat nat, R == R.
```

This term does not type-check since Coq is unable to unify `rel nat nat` (the declared type for `R`) with `@X _ _ _` (the type which is expected on both sides of a `==`). A solution in this case consists in declaring the instance `rel_G` from Fig. 5 as a "canonical structure": doing so precisely tells Coq to use `rel_G` when facing such a unification problem. (By the way, this also tells Coq to use `rel_G` for unification problems of the form `Type` $=_\beta$ `@T _`, which is required by the above example as well.)

Unfortunately, this trick does not play well with our peculiar representation of untyped models, like languages or regular expressions (Fig. 5 and 7). Indeed, the dummy occurrences of `unit` parameters prevent Coq from using the instance `lang_G` as a canonical structure. Our solution in this case consists in using an appropriate notation to hide the corresponding occurrences of `X` behind an informative name:

```
Notation language := (@X lang_G tt tt).
Notation regex := (@X re_G tt tt).
Check forall L: language, L⋆ · L⋆ == L⋆.
```

Also note that the ability to declare more general hints for unification [15] would certainly help to solve this problem in a nicer way.

2.4.3. *Separation between operations an laws.* When defining the classes for the algebraic structures, it might seem more natural to package operations together with their laws. For example, we could merge the classes `Monoid_Ops` and `Monoid` from Fig. 3 and 4. There are at least two reasons for keeping separate classes.

First, by separating operational contents from proof contents, we avoid the standard problems due to the lack of proof irrelevance in Coq, and situations where typeclass resolution might be ambiguous. Indeed, having two proofs asserting that some operations form a semiring is generally harmless; however, if we pack operations with the proof that they satisfy some laws, then two distinct proofs sometimes mean two different operations, which becomes highly problematic. This would typically forbid the technique we presented above to factorise some proofs by duality.

Second, this makes it possible to define other structures sharing the same operations (and hence, notations), but not necessarily the same laws. We exploit this possibility, for example, to define a class for Kleene algebra *with converse* using fewer laws: the good properties of the converse operation provide more symmetries so that some laws become redundant (we use this class to get shorter proofs for the instances from Fig. 5: the models of binary relations and languages both have a converse operation).

This choice is not critical for the library in its current state, because we basically stop at Kleene algebra. However, based on preliminary experiments, having this separation is crucial when considering richer structures like residuated Kleene lattices [35] or allegories [24].

3. Matrices.

In this section, we describe our implementation of matrices, building on the previously described framework. Matrices are indeed required to formalise Kozen's initiality proof [38], as explained in Sect. 1.2.

3.1. **Which matrices to construct?**

Assume a graph `G`. There are at least three ways of defining a new graph for matrices:

(1) Fix an object $\texttt{u} \in \texttt{T}$ and use natural numbers ($\mathbb{N}$) as objects: morphisms between n and m are $n \times m$ matrices whose elements belong to the square homset `X u u`.
(2) Use pairs $(\texttt{u}, n) \in \texttt{T} \times \mathbb{N}$ as objects: morphisms from $(\texttt{u}, n)$ to $(\texttt{v}, m)$ are $n \times m$ matrices with elements in `X u v`.
(3) Use lists $[\texttt{u}_1, \dots, \texttt{u}_n] \in \texttt{T}^\star$ as objects: a morphism from $[\texttt{u}_1, \dots, \texttt{u}_n]$ to $[\texttt{v}_1, \dots, \texttt{v}_m]$ is an $n \times m$ matrix M such that $M_{i,j}$ belongs to $\texttt{X}\ \texttt{u}_i\ \texttt{v}_j$.

The third option is the most theoretically appealing one: this is the most general construction. Although we can actually build a typed Kleene algebra of matrices in this way, this requires dealing with a lot of dependent types, which can be tricky. The second option is also rather natural from the mathematical point of view and it does not impose a strongly dependent typing discipline.

However, while formalising the second or the third option is interesting *per se*, to get new models of typed Kleene algebras, the first construction actually suffices for Kozen's initiality proof. Indeed, this proof only requires matrices over regular expressions and languages. Since these two models are untyped (their type `T` for objects is just `unit`), the three possibilities coincide (we can take `tt` for the fixed object `u` without loss of generality). In the end, we chose the first option, because it is the simplest one.

3.2. **Coq representation for matrices.**

According to the previous discussion, we assume a graph `G: Graph` and an object `u: T`. We furthermore abbreviate the type `X u u` as `X`: this is the type of the elements—sometimes called scalars.

```
Context {SLo: SemiLattice_Ops G}.
Fixpoint sum i k (f: nat → X) :=
  match k with
  | O ⇒ O
  | S k ⇒ f i + sum (S i) k f
  end.
```

```
Context {Mo: Monoid_Ops G}.
Definition mx_dot n m p (M: MX n m) (N: MX m p) :=
  fun i j ⇒ sum O m (fun k ⇒ M i k · N k j).

Definition mx_one n: MX n n :=
  fun i j ⇒ if eq_nat_bool i j then 1 else 0.
```

Figure 10. Definition of matricial product and identity matrix.

Dependently typed representation. A matrix can be seen as a partial map from pairs of integers to X, so that the Coq type for matrices could be defined as follows:

```
Definition MX (n m: nat) := forall i j, i<n → j<m → X.
Definition mx_equal n m (M N: MX n m) i j (Hi: i<n) (Hj: j<m) := M i j Hi Hj == N i j Hi Hj.
```

This corresponds to the dependent types approach: a matrix is a map to X from two integers and two proofs that these integers are lower than the bounds of the matrix. Except for the concrete representation, this is the approach followed in [5, 25, 7]. With such a type, every access to a matrix element is made by exhibiting two proofs, to ensure that indices lie within the bounds. This is not problematic for simple operations like the function mx_plus below: it suffices to pass the proofs around; this however requires more boilerplate for other functions, like block decomposition operations.

```
Context {SLo: SemiLattice_Ops G}.
Definition mx_plus n m (M N: MX n m) i j (Hi: i<n) (Hj: j<m) := M i j Hi Hj + N i j Hi Hj.
```

Infinite functions. We actually adopt another strategy: we move bounds checks to equality proofs, by working with the following definitions:

```
Definition MX n m := nat → nat → X.
Definition mx_equal n m (M N: MX n m) := forall i j, i<n → j<m → M i j == N i j.
```

Here, a matrix is an infinite function from pairs of integers to X, only equality is restricted to the actual domain of the matrix. With these definitions, we do not need to manipulate proofs when defining matrix operations, so that subsequent definitions are easier to write. For instance, the functions for matrix multiplication and block manipulations are given in Fig. 10 and Fig. 11. For multiplication, we use a very naive function to compute the appropriate sum: there is no need to provide an explicit proof that each call to the functional argument is performed within the bounds.

Similarly, the mx_sub function, for extracting a sub-matrix, has a very liberal type: it takes an arbitrary $p \times q$ matrix M, it returns an arbitrary $n \times m$ matrix, and this matrix is obtained by reading M from an arbitrary position (x, y). This function is then instantiated with more sensible arguments to get the four functions corresponding to the decomposition of an $(x + n) \times (y + m)$ matrix into four blocks. The converse function, to define a matrix by blocks, is named mx_blocks.

Bounds checks are required a posteriori only, when proving properties about these matrix operations, e.g., that multiplication is associative or that the four sub-matrix functions preserve matricial equality. This is generally straightforward: these proofs are done within the interactive proof mode, so that bound checks can be proved with high-level tactics like

```
Definition mx_sub p q x y n m
  (M: MX p q): MX n m :=
  fun i j ⇒ M (x + i) (y + j).

Variables x y n m: nat.
Definition mx_sub00 := mx_sub (x+n) (y+m) 0 0 x y.
Definition mx_sub01 := mx_sub (x+n) (y+m) 0 y x m.
Definition mx_sub10 := mx_sub (x+n) (y+m) x 0 n y.
Definition mx_sub11 := mx_sub (x+n) (y+m) x y n m.
```

```
Definition mx_blocks x y n m
  (M: MX x y) (N: MX x m)
  (P: MX n y) (Q: MX n m): MX (x+n) (y+m)
  := fun i j ⇒ match S i−x, S j−y with
      | O,   O   ⇒ M i j
      | O,   S j ⇒ N i j
      | S i, O   ⇒ P i j
      | S i, S j ⇒ Q i j
     end.
```

Figure 11. Definition of sub-matrix extraction and block matrix construction.

omega. (Note that a similar behaviour could also be achieved with a dependently typed definition of matrices by using Coq's Program feature. We prefer our approach for its simplicity: Program tends to generate large terms which are not so easy to work with.)

The correctness proof of our algorithm heavily relies on matricial reasoning (Sect. 4), and in particular block matrix decomposition (Sect. 3.3 and 4.2). Despite this fact, we have not found major drawbacks to this approach yet. We actually believe that it would scale smoothly to even more intensive usages of matrices like, e.g., linear algebra [27].

Phantom types. Unfortunately, these non-dependent definitions allow one to type the following code, where the three additional arguments of `dot` are implicit:

```
Definition ill_dot n p (M: MX n 16) (N: MX 64 p): MX n p := dot M N.
```

This definition is accepted thanks to the conversion rule: the dependent type `MX n m` does not mention `n` nor `m` in its body, so that these arguments can be discarded by the type system (we actually have `MX n 16` = `MX n 64`). While such an ill-formed definition will be detected at proof-time; it is a bit sad to loose the advantages of a strongly typed programming language here. We solved this problem at the cost of some syntactic sugar, by resorting to an inductive singleton definition, reifying bounds in *phantom types*:

```
Inductive MX (n m: nat) := box: (nat → nat → X) → MX n m.
Definition get n m (M: MX n m) := match M with box f ⇒ f end.
Definition mx_plus n m (M N: MX n m) := box n m (fun i j ⇒ get M i j  + get N i j).
```

Coq no longer equates types `MX n 16` and `MX n 64` with this definition, so that the above `ill_dot` function is rejected, and we can trust inferred implicit arguments (e.g., the `m` argument of `dot`).

Computation. Although we do not use matrices for computations in this work, we also advocate this lightweight representation from the efficiency point of view. First, using non-dependent types is more efficient: not a single boundary proof gets evaluated in matrix computations. Second, using functions to represent matrices allows for fine-grain optimisation: it gives a lazy evaluation strategy by default, which can be efficient if the matrix resulting of a computation is seldom used, but we can also enforce a call-by-value behaviour for some expressions, to avoid repeating numerous calls to a given expensive computation.

Indeed, we can define a *memoisation* operator that computes all elements of a given matrix, stores the results in a map, and returns the closure that looks up in the map rather than recomputing the result. The map can be implemented using lists or binary trees, for example. In any case, we can then prove this memoisation operator to be an identity so that it can be inserted in matrix computations in a transparent way, at judicious places.

```
Definition mx_force n m (M: MX n m): MX n m :=
  let l := mx_to_maps M in box n m (fun i j ⇒ mget i (mget j l)).
Lemma mx_force_id : forall n m (M : MX n m), mx_force M == M.
```

3.3. **Taking the star of a matrix.** As expected, we declare the previous operations on matrices (e.g., Fig. 10) as new instances, so that we can directly use notations, lemmas, and tactics with matrices. The type of these instances are given below:

```
Instance mx_G: Graph := { T := nat; X := MX; equal := mx_equal }.

Instance mx_SLo: SemiLattice_Ops G → SemiLattice_Ops mx_G.
Instance mx_Mo:  SemiLattice_Ops G → Monoid_Ops G → Monoid_Ops mx_G.
Instance mx_Ko:  SemiLattice_Ops G → Monoid_Ops G → Star_Op G → Star_Op mx_G.

Instance mx_SL:  `{SemiLattice G} → SemiLattice mx_G.
Instance mx_ISR: `{IdemSemiRing G} → IdemSemiRing mx_G.
Instance mx_KA:  `{KleeneAlgebra G} → KleeneAlgebra mx_G.
```

To obtain the fourth and last instances, we have to define a star operation on matrices, and show that it satisfies the laws for Kleene star. We conclude this section about matrices by a brief description of this construction—see [38] for a detailed proof.

The idea is to proceed by induction on the size of the matrix: the problem is trivial if the matrix is empty or of size 1×1; otherwise, we decompose the matrix into four blocks and we recurse as follows [1]:

$$\left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right]^\star = \left[\begin{array}{c|c} A' & A' \cdot B \cdot D' \\ \hline D' \cdot C \cdot A' & D' + D' \cdot C \cdot A' \cdot B \cdot D' \end{array}\right] \quad \text{where} \begin{cases} D' = D^\star, \\ A' = (A + B \cdot D' \cdot C)^\star \end{cases} \qquad (\dagger)$$

This definition may look mysterious; the special case where C is zero might be more intuitive:

$$\left[\begin{array}{c|c} A & B \\ \hline 0 & D \end{array}\right]^\star = \left[\begin{array}{c|c} A^\star & A^\star \cdot B \cdot D^\star \\ \hline 0 & D^\star \end{array}\right] . \qquad (\ddagger)$$

As long as we take square matrices for A and D, the way we decompose the matrix does not matter (we actually have to prove it). In practice, since we work with Coq natural numbers (`nat`), we choose A of size 1×1: this allows recursion to go smoothly (if we were interested in efficient matrix computations, it would be better to half the matrix size).

The corresponding code is given in Fig. 12. We first define an auxiliary function, `mx_star`', which follows the above definition by blocks (†), assuming two functions to perform the recursive calls (i.e., to compute A' and D'). The function `mx_star_11` computes the star of a 1×1 matrix by using the star operation on the underlying element. Using these two functions, we get the final `mx_star` function as a simple fixpoint. The proof that this operation satisfies the laws of Kleene algebras is complicated [38]; note that by making explicit the general block definition with the auxiliary function `mx_star`', we can easily state theorem `mx_star_block`: equation (†) holds for each possible decomposition of the matrix.

```
Definition mx_star' x n
  (sx: MX x x → MX x x)
  (sn: MX n n → MX n n)
  (M: MX (x+n) (x+n)): MX (x+n) (x+n) :=
 let A := mx_sub00 M in
 let B := mx_sub01 M in
 let C := mx_sub10 M in
 let D := mx_sub11 M in
 let D' := sn D in
 let A' := sx (A + B · D' · C) in
   mx_blocks
        A'          (A' · B · D')
     (D' · C · A') (D' + D' · C · A' · B · D').
```

```
Definition mx_star_11 (M: MX 1 1): MX 1 1 :=
  fun _ _ ⇒ (M 0 0)⋆.

Fixpoint mx_star n: MX n n → MX n n :=
  match n with
  | 0 ⇒ fun M ⇒ M
  | S n ⇒ mx_star' mx_star_11 (mx_star n)
  end.

Theorem mx_star_block x n (M: MX (x+n) (x+n)):
  mx_star (x+n) M ==
    mx_star' (mx_star x) (mx_star n) M.
Proof...
```

Figure 12. Definition of the star operation on matrices.

4. The algorithm and its proof

We now focus on the heart of our tactic: the decision procedure and the corresponding correctness proof. The algorithm we chose to implement to decide whether two regular expressions denote the same language can be decomposed into five steps:

(1) normalise both expressions to turn them into "strict star form";
(2) build non-deterministic finite automata with epsilon-transitions (ϵ-NFA);
(3) remove epsilon-transitions to get non-deterministic finite automata (NFA);
(4) determinise the automata to obtain deterministic finite automata (DFA);
(5) check that the two DFAs are equivalent.

The fourth step can produce automata of exponential size. Therefore, we have to carefully select our construction algorithm, so that it produces rather small automata. More generally, we have to take a particular care about efficiency; this drives our choices about both data structures and algorithms.

The Coq types we used to represent finite automata are given in Fig. 13; we use modules only for handling the name-space; the type `regex` is that from Fig. 7 (Sect. 2.3), `label` and `state` are aliases for the type of numbers. The first record type, `MAUT.t`, corresponds to the matricial representation of automata; it is rather high-level but computationally inefficient (`MX n m` is the type of $n \times m$ matrices over `regex`—Sect. 3). We only use this type in proofs, through the evaluation function `MAUT.eval` (the function `mx_to_scal` casts a 1×1 matrix into a regular expression). The three other types are efficient representations for the three kinds of automata we mentioned above; fields `size` and `labels` respectively code for the number of states and labels, the other fields are self-explanatory. In each case, we define a translation function to matricial automata, `to_MAUT`, so that each kind of automata can eventually be evaluated into a regular expression.

The overall structure of the correctness proof is depicted in Fig. 14. Datatypes are recalled on the left-hand side; the outer part of the right-hand side corresponds to computations: starting from two regular expressions x and y, two DFAs A_3 and B_3 are constructed

```
Module MAUT.
 Record t := mk {
  size:     nat;
  initial:  MX 1 size;
  delta:    MX size size;
  final:    MX size 1 }.
 Definition eval(A: t): regex :=
  mx_to_scal (initial A · delta A⋆ · final A).
End MAUT.
```

```
Module eNFA.
 Record t := mk {
  size:     state;
  labels:   label;
  epsilon:  state → stateset;
  delta:    label → state → stateset;
  initial:  state;
  final:    state }.
 Definition to_MAUT(A: t): MAUT.t := ...
 Definition eval := MAUT.eval ∘ to_MAUT.
End eNFA.
```

```
Module NFA.
 Record t := mk {
  size:     state;
  labels:   label;
  delta:    label → state → stateset;
  initial:  stateset;
  final:    stateset }.
 Definition to_MAUT(A: t): MAUT.t := ...
 Definition eval := MAUT.eval ∘ to_MAUT.
End NFA.
```

```
Module DFA.
 Record t := mk {
  size:     state;
  labels:   label;
  delta:    label → state → state;
  initial:  state;
  final:    stateset }.
 Definition to_MAUT(A: t): MAUT.t := ...
 Definition eval := MAUT.eval ∘ to_MAUT.
End DFA.
```

Figure 13. Coq types and evaluation functions for the four automata representations.

and tested for equivalence. The proof corresponds to the inner equalities (`==`): each automata construction preserves the semantics of the initial regular expressions, two DFAs evaluate to equal values when they are declared equivalent by the corresponding algorithm.

In the following sections, we give more details about each step of the decision procedure, together with a sketch of our correctness proof (although we work with different algorithms, this proof is largely based on Kozen's one [38]).

4.1. **Normalisation, strict star form.** There exists no complete rewriting system to decide equations of Kleene algebra (their equational theory is not finitely based [50]); this is why one usually goes through finite automata constructions. One can still use rewriting techniques to simplify the regular expressions before going into these expensive constructions. By doing so, one can reduce the size of the generated automata, and hence, the time needed to check for their equivalence.

For example, a possibility consists in normalising expressions with respect to the following convergent rewriting system. (Although we actually implemented this trivial optimisation, we will not discuss it here.)

$$\begin{array}{cccc} x\cdot 0\to 0 & 0\cdot x\to 0 & x+0\to x & 0+x\to x \\ x\cdot 1\to x & 1\cdot x\to x & 0^\star\to 1 & \end{array}$$

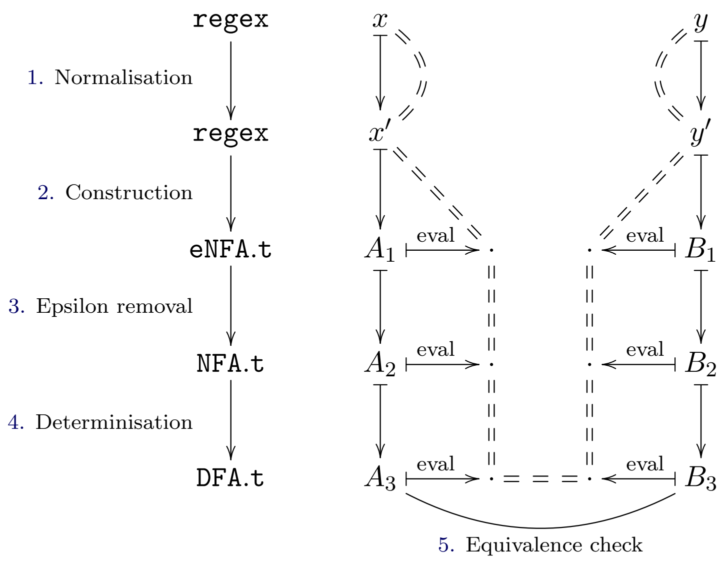


Figure 14. Overall picture for the algorithm and its correctness.

Among other laws one might want to exploit in a preliminary normalisation step, there are the following ones:

$$1^\star \to 1 \qquad\qquad x^{\star\star} \to x^\star \ .$$

More generally, any star expression $x^\star$ where x accepts the empty word can be simplified using the simple syntactic procedure proposed by Brüggemann-Klein [12]. For example, this procedure reduces the expression on the left-hand side below to the one on the right-hand side, which is in *strict star form*: all occurrences of the star operation act on *strict* regular expressions, regular expressions that do not accept the empty word.

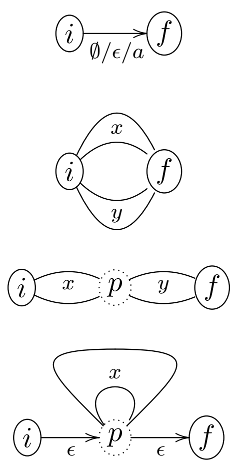
$$((a+1)\cdot((b+1)^\star\cdot c+d^\star))^\star \quad\to\quad (a+b^\star\cdot c+d)^\star \ .$$

In Coq, this procedure translates into a simple fixpoint whose correctness relies on the following laws:

$$\begin{aligned}(x+1)^\star &= x^\star\\ (x+y^\star)^\star &= (x+y)^\star\\ (x\cdot y)^\star &= (x+y)^\star \qquad \text{(if } x \text{ and } y \text{ accept the empty word)}\end{aligned}$$

```
Fixpoint ssf: regex → regex := ...
Theorem ssf_correct: forall x, ssf x == x.
```

The above theorem corresponds to the first step of the overall proof, as depicted in Fig. 14. As we shall explain in Sect. 4.3, working with expressions in strict star form also allows us to get a simpler and more efficient algorithm to remove epsilon transitions. This means that we also proved the `ssf` function complete, i.e., that it always produces expressions in strict star form:



```
Fixpoint build x i f A :=
  match x with
    | zero    ⇒ A
    | one     ⇒ add_one i f A
    | var a   ⇒ add_var a i f A
    | plus x y ⇒ build x i f (build y i f A)
    | dot x y  ⇒
        let (p,A) := incr A in
          build x i p (build y p f A)
    | star x   ⇒
        let (p,A) := incr A in
          add_one i p (build x p p (add_one p f A))
  end.
```

Figure 15. Construction algorithm—a variant of Ilie and Yu's construction.

```
Inductive strict_star_form: regex → Prop := ...
Theorem ssf_complete: forall x, strict_star_form (ssf x).
```

One could also normalise expressions modulo idempotence of $+$, to avoid replications in the generated automata. This in turn requires normalising terms modulo associativity and commutativity of $+$, and associativity of $\cdot$, so that terms like $((a+b)\cdot c)\cdot d+(b+a)\cdot(c\cdot d)$ can be reduced modulo idempotence. Such a phase can easily be implemented, but it results in a slower procedure in practice (normalisation requires quadratic time and non-trivial instances of the idempotence law do not appear so frequently). We do not include this step in the current release.

4.2. **Construction.** There are several ways of constructing an ϵ-NFA from a regular expression. At first, we implemented Thompson's construction [56], for its simplicity; we finally switched to a variant of Ilie and Yu's construction [34], which produces smaller automata. This algorithm constructs an automaton with a single initial state and a single accepting state (respectively denoted by i and f); it proceeds by structural induction on the given regular expression. The corresponding steps are depicted on the left-hand side of Fig. 15; the first drawing corresponds to the base cases (zero, one, variable); the second one is union (plus): we recursively build the two sub-automata between i and f; the third one is concatenation: we introduce a new state, p, build the first sub-automaton between i and p, and the second one between p and f; the last one is for iteration (star): we build the sub-automata between a new state p and p itself, and we link i, p, and f with two epsilon-transitions. The corresponding Coq code is given on the right-hand side. To avoid costly union operations, we actually use an accumulator (`A`) to which we recursively add states and transitions (the functions `add_one` and `add_var` respectively add epsilon and labelled transitions to the accumulator—the function `incr` adds a new state to the accumulator and returns this state together with the extended accumulator).

We actually implemented this algorithm twice, by using two distinct datatypes for the accumulator: first, with a high-level matricial representation; then with efficient maps for storing epsilon and labelled transitions. Doing so allows us to separate the correctness

```
Module pre_MAUT.
 Record t := mk {
  size:  nat;
  delta: MX size size }.

 Definition to_MAUT i f A := MAUT.mk
  (mx_point 0 i 1) (delta A) (mx_point f 0 1).
 Definition eval i f := MAUT.eval ∘ (to_MAUT i f)

 Definition add (x: regex) i f A :=
  mk _ (delta A  + mx_point i f x)
 Definition add_one := add 1.
 Definition add_var a := add (var a).
 Definition incr A := let mk n M := A in
  (n, mk (n + 1) (mx_blocks M 0 0 0)).
 Fixpoint build x i f A := (* Fig. 15 *).

 Definition empty := mk 2 0.
 Definition regex_to_MAUT x :=
  to_MAUT 0 1 (build x 0 1 empty).
End pre_MAUT.
```

```
Module pre_eNFA.
 Record t := mk {
  size:     state;
  labels:   label;
  epsmap:   statemap stateset;
  deltamap: statelabelmap stateset }.

 Definition to_eNFA i f A := ...



 Definition add_one i f A := ...
 Definition add_var a i f A := ...
 Definition incr A := ...

 Fixpoint build x := (* Fig. 15 *).

 Definition empty := mk 2 0 [] [].
 Definition regex_to_eNFA x :=
  to_eNFA 0 1 (build x 0 1 empty).
End pre_eNFA.
```

Figure 16. The two modules for the construction algorithm.

proof into an algebraic part, which we can do with the high-level representation, and an implementation-dependent part consisting in showing that the two versions are equivalent.

These two versions correspond to the modules given in Fig. 16. Basically, we have the record types `MAUT.t` and `eNFA.t` from Fig. 13, without the fields for initial and final states. (The other difference being that we use maps rather than functions on the the efficient side—`pre_eNFA`.) On the high-level side—`pre_MAUT`, we use generic matricial constructions: adding a transition to the automaton consists in performing an addition with the matrix containing only that transition (`mx_point i f x` is the matrix with `x` at position `(i,f)` and zeros everywhere else); adding a state to the automaton consists in adding a empty row and a empty column to the matrix, thanks to the `mx_blocks` function (defined in Fig. 11). We did not include the corresponding details for the low-level representation: they are slightly verbose and they can easily be deduced. Notice that `pre_NFA` does not include a generic `add` function: while the matricial representation allows us to label transitions with arbitrary regular expressions, the efficient representation statically ensures that transitions are labelled either with epsilon or with a variable (a letter of the alphabet).

The final construction functions, from `regex` to `MAUT.t` or `eNFA.t`, are obtained by calling `build` between the two states 0 and 1 of an empty accumulator (note that the occurrence of 0 in the definition of `pre_MAUT.empty` denotes the empty $(2,2)$-matrix).

Since the two versions of the algorithm only differ by their underlying data structures, proving that they are equivalent is routine ([=] denotes matricial automata equality):

```
Lemma constructions_equiv: forall x, regex_to_MAUT x [=] eNFA.to_MAUT (regex_to_eNFA x).
```

Let us now focus on the algebraic part of the proof. We have to show:

```
Theorem construction_correct: forall x, MAUT.eval (regex_to_MAUT x) == x.
```

The key lemma is the following one: calling `build x i f A` to insert an automaton for the regular expression `x` between the states `i` and `f` of `A` is equivalent to inserting directly a transition with label `x` (recall that transitions can be labelled with arbitrary regular expressions in matricial automata); moreover, this holds whatever the initial and final states `s` and `t` we choose for evaluating the automaton.

```
Lemma build_correct: forall x i f s t A,
  i<size A → f<size A → s<size A → t<size A →
  eval s t (build x i f A) == eval s t (add x i f A).
```

As expected, we proceed by structural induction on the regular expression `x`. As an example of the involved algebraic reasoning, the following property of star w.r.t. block matrices is used twice in the proof of the above lemma: with $(x, y, z) = (e, 0, f)$, it gives the case of a concatenation $(e \cdot f)$; with $(x, y, z) = (1, e, 1)$ it yields iteration $(e^\star)$. This laws follows from the general characterisation of the star operation on block matrices (Equation (†) in Sect. 3.3). In both cases, the line and the column that are added on the left-hand side correspond to the state (p) generated by the construction.

$$\left[\begin{array}{c|c} \quad u \quad & 0 \end{array}\right] \cdot \left[\begin{array}{ccc|c} & \vdots & & 0 \\ \cdots & M_{i,f} & \cdots & x \\ & \vdots & & 0 \\ \hline 0 & z & 0 & y \end{array}\right]^\star \cdot \left[\begin{array}{c} v \\ \\ \hline 0 \end{array}\right] = u \cdot \left[\begin{array}{ccc} & \vdots & \\ \cdots & M_{i,f} + x \cdot y^\star \cdot z & \cdots \\ & \vdots & \end{array}\right]^\star \cdot v$$

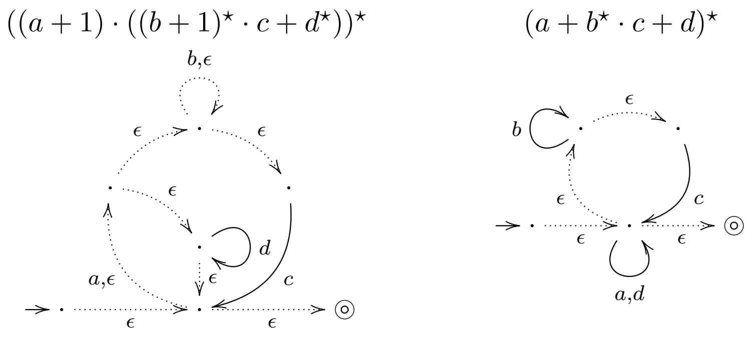
In the special case where `A` is the empty accumulator, lemma `build_correct` gives:

$$\begin{array}{rcl} \texttt{MAUT.eval (regex\_to\_MAUT x)} & \texttt{==} & \texttt{eval 0 1 (build x 0 1 empty)} \\ & \texttt{==} & \texttt{eval 0 1 (add x 0 1 empty)} \\ & \texttt{==} & \begin{bmatrix} 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 & x \\ 0 & 0 \end{bmatrix}^\star \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\ & \texttt{==} & \begin{bmatrix} 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & x \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\ & \texttt{==} & \texttt{x} \end{array}$$

i.e., theorem `construction_correct`.
Finally, by combining the equivalence of the two algorithms (lemma `constructions_equiv`) and the correctness of the high-level one (theorem `construction_correct`), we obtain the correctness of the efficient construction algorithm. In other words, we can fill the two triangles corresponding to the second step in Fig. 14:

```
Theorem construction_correct': forall x, eNFA.eval (regex_to_eNFA x) == x.
```

4.3. **Epsilon transitions removal.** The automata obtained with the above construction contain epsilon-transitions: each starred sub-expression produces two epsilon-transitions, and each occurrence of 1 gives one epsilon-transition. Indeed, their transitions matrices are of the form $M = J + N$ with $N = \sum_a a \cdot N_a$, where J and the N_a are 0-1 matrices. These matrices just correspond to the graphs of epsilon and labelled transitions.



$((a+1)\cdot((b+1)^\star\cdot c+d^\star))^\star$
$(a+b^\star\cdot c+d)^\star$
b,ϵ
ϵ
a,ϵ
d
c
b
a,d

Figure 17. Running the construction algorithm on an expression and its strict star form.

Removing epsilon-transitions can be done at the algebraic level using the following law:

$$(x+y)^\star = x^\star \cdot (y \cdot x^\star)^\star \quad ,$$

from which we get

$$u \cdot (J+N)^\star \cdot v = u \cdot J^\star \cdot (N \cdot J^\star)^\star \cdot v \quad ,$$

so that the automata $\langle u, M, v\rangle$ and $\langle u \cdot J^\star, N \cdot J^\star, v\rangle$ are equivalent. We can moreover notice that the latter automaton no longer contains epsilon-transitions: this is a NFA (the transition matrix, $N \cdot J^\star$, can be written as $\sum_a a \cdot N_a \cdot J^\star$, where the $N_a \cdot J^\star$ are 0-1 matrices).

This algebraic proof is not surprising: looking at 0-1 matrices as binary relations between states, $J^\star$ actually corresponds to the reflexive-transitive closure of J.

Although this is how we prove the correctness of this step, computing $J^\star$ algebraically is inefficient: we have to implement a proper transitive closure algorithm for the low-level representation of automata. We actually rely on a property of the construction from Sect. 4.2: when given regular expressions in strict star form (Sect. 4.1), the produced ϵ-NFAs have acyclic epsilon-transitions. Intuitively, the only possibility for introducing an epsilon-cycle in the construction from Sect. 4.2 comes from star expressions. Therefore, by forbidding the empty word to appear in such cases, we prevent the formation of epsilon-cycles.

Consider for example Fig. 17, where we have executed the construction algorithm of Fig. 15 on two regular expressions (these are the expressions from Sect. 4.1—the right-hand side expression is the strict star form of the left-hand side one). There are two epsilon-loops in the left hand-side automaton, corresponding to the two occurrences of star that are applied to non-strict expressions ($(b+1)^\star$ and the whole term). On the contrary, in the automaton generated from the strict star form—the second regular expression, the states belonging to these loops are merged and the corresponding transitions are absent: the epsilon-transitions form a directed acyclic graph (here, a tree).

This acyclicity property allows us to use a very simple algorithm to compute the transitive closure. With respect to standard algorithms for the general (cyclic) case, this algorithm is easier to implement in Coq, slightly more efficient, and simpler to certify. More concretely, we need to prove that the construction algorithm returns ϵ-NFAs whose reversed epsilon-transitions are well-founded, when given expressions in strict star form:

```
Definition eNFA_well_founded A :=
  well_founded (fun i j ⇒ In i (eNFA.epsilon A j)).
```

```
Theorem construction_wf: forall x,
  strict_star_form x → eNFA_well_founded (regex_to_eNFA x).
```

(Note that this proof is non-trivial.) Our function to convert ϵ-NFAs into NFAs takes such a well-founded proof as an argument, and uses it to compute the reflexive-transitive closure of epsilon-transitions:

```
Definition eNFA_to_NFA (A: eNFA.t): eNFA_well_founded A → NFA.t := ...
```

This step is easy to implement since we can proceed by well-founded induction. In particular, there is no need to bound the recursion level with the number of states, to keep track of the states whose transitive closure is being computed to avoid infinite loops, or to prove that a function defined in this way terminates. Note that we still use memoisation, to take advantage of the sharing offered by the directed acyclic graph structure. Also note that since this function hasto be executed efficiently, we use a standard Coq trick by Bruno Barras to avoid the evaluation of the well-foundness proof: we guard this proof with a large amount of constructors so that the actual proof is never reached in practice.

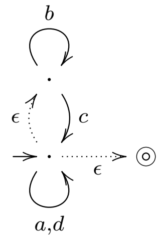
We finally prove that the previous function returns an automaton whose translation into a matricial automaton is exactly $\langle u \cdot J^\star, N \cdot J^\star, v\rangle$, so that the above algebraic proof applies. This closes the third step in Fig. 14.

```
Theorem epsilon_correct: forall A (HA: eNFA_well_founded A),
  NFA.eval (eNFA_to_NFA A HA) == eNFA.eval A.
```

Comparison with Ilie and Yu's construction. Let us make a digression here to compare our construction algorithm with the one proposed by Ilie and Yu [34, Algorithm 4, p. 144]. The steps of the recursive procedure, as presented in Fig. 15, are exactly the same; the only difference is that they refine the automaton by merging some states and removing useless transitions:

(a) the state introduced in the `dot` case is removed when it is preceded or followed by a single epsilon-transition;
(b) epsilon-cycles introduced in the `star` case are merged into a single state;
(c) if at the end of the algorithm, the initial state only has one outgoing epsilon-transition, the initial state is shifted along this transition;
(d) duplicated transitions are merged into a single one.

For instance, running Ilie and Yu's construction on the right-hand side expression of Fig. 17 yields the automaton on the right. This automaton is actually smaller than the one we generate: two states and two epsilon-transitions are removed using (a) and (c). Moreover, thanks to optimisation (b), Ilie and Yu also get this automaton when starting from the left-hand side expression, although this expression is not in strict star form.



We did not implement (a) for two reasons: first, this optimisation is not so simple to code efficiently (we need to be able to merge states and to detect that only one epsilon transition reaches a given state), second, it was technically involved to prove its correctness at the algebraic level (recall that we need to motivate each step by some matricial reasoning). Similarly, although step (c) is easy to implement, proving its correctness would require substantial additional work. On the contrary, our presentation of the algorithm directly enforces (d): the data structures we use systematically merge duplicate transitions.

The remaining optimisation is (b), which would be even harder to implement and to prove correct than (a). Fortunately, by working with expressions in strict star form, the need for this optimisation vanishes: epsilon-cycles cannot appear. In the end, although we implement (b) by putting expressions in strict star form first, the only difference with Ilie and Yu's construction is that we do not perform steps (a) and (c).

4.4. **Determinisation.** Starting from a NFA $\langle u, M, v\rangle$ with n states, the determinisation algorithm consists in a standard depth-first enumeration of the subsets that are accessible from the set of initial states. It returns a DFA $\langle \widehat{u}, \widehat{M}, \widehat{v}\rangle$ with $\widehat{n}$ states, together with a injective map ρ from $[1..\widehat{n}]$ to subsets of $[1..n]$. We sketch the algebraic part of the correctness proof. Let X be the rectangular $(\widehat{n}, n)$ 0-1 matrix defined by $X_{sj} \triangleq j \in \rho(s)$; the intuition is that X is a "decoding" matrix: it sends states of the DFA to the characteristic vectors of the corresponding subsets of the NFA. By a precise analysis of the algorithm, we prove that the following commutation properties hold:

$$\widehat{M} \cdot X = X \cdot M \quad (1) \qquad\qquad \widehat{u} \cdot X = u \quad (2) \qquad\qquad \widehat{v} = X \cdot v \quad (3)$$

Equation (1) can be read as follows: executing a transition in the DFA and then decoding the result is equivalent to decoding the starting state and executing parallel transitions in the NFA. Similarly, (2) states that the initial state of the DFA corresponds to the set of initial states of the NFA, and (3) assesses that the final states of the DFA are those containing at least one accepting state of the NFA.

From (1), we deduce that $\widehat{M}^{\star} \cdot X = X \cdot M^{\star}$ using the lemma `iter` from Sect. 2.4.1; we conclude with (2, 3):

$$\widehat{u} \cdot \widehat{M}^{\star} \cdot \widehat{v} = \widehat{u} \cdot \widehat{M}^{\star} \cdot X \cdot v = \widehat{u} \cdot X \cdot M^{\star} \cdot v = u \cdot M^{\star} \cdot v \ .$$

The DFA evaluates like the starting NFA: we can fill the two squares corresponding to the fourth step in Fig. 14.

Let us mention a Coq-specific technical difficulty in the concrete implementation of this algorithm. The problem comes from termination: even though it theoretically suffices to execute the main loop at most 2^n times (there are 2^n subsets of $[1..n]$), we cannot use this bound directly in practice. Indeed, NFAs with 500 states frequently result in DFAs of about a thousand states, which we should be able to compute easily. However, using the number 2^n to bound the recursion depth in Coq requires to compute this number before entering the recursive function. For $n = 500$ this is obviously out of reach (this number has to be in unary format—`nat`—since it is used to ensure structural recursion).

We have tried to use well-founded recursion, which was rather inconvenient: this requires mixing some non-trivial proofs with the code. We currently use the following "pseudo-fixpoint operators", defined in continuation passing style:

```
Variables A B: Type.
Fixpoint linearfix n (f: (A → B) → A → B) (k: A → B) (a: A): B :=
  match n with O ⇒ k a | S n ⇒ f (linearfix n f k) a end.
Fixpoint powerfix n (f: (A → B) → A → B) (k: A → B) (a: A): B :=
  match n with O ⇒ k a | S n ⇒ f (powerfix n f (powerfix n f k)) a end.
```

Intuitively, `linearfix n f k` lazily approximates a potential fixpoint of the functional `f`: if a fixpoint is not reached after `n` iterations, it uses `k` to escape. The `powerfix` operator behaves similarly, except that it escapes after $2^n - 1$ iterations: we prove that `powerfix n`

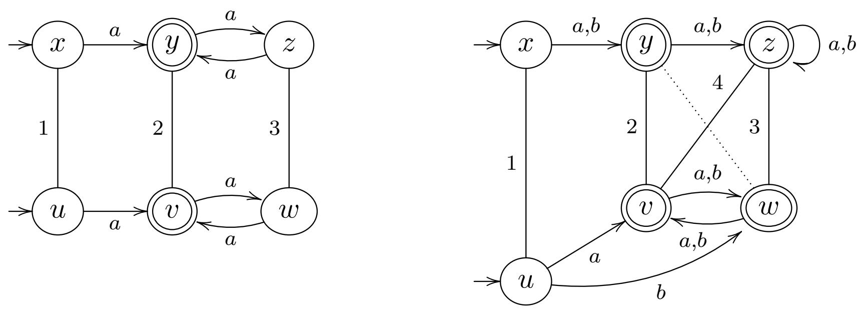


Figure 18. Checking for DFA equivalence (Hopcroft and Karp).

`f k a` is equal to `linearfix` $(2^n - 1)$ `f k a`. Thanks to these operators, we can write the code to be executed using `powerfix`, while keeping the ability to reason about the simpler code obtained with a naive structural iteration over 2^n: both versions of the code are easily proved equivalent, using the intermediate `linearfix` characterisation.

4.5. **Equivalence checking.** Two DFAs are equivalent if and only if their respective minimised DFAs are equal up-to isomorphism. Therefore, computing the minimised DFAs and exploring all state permutations is sufficient to obtain decidability.

However, there is a more direct and efficient approach that does not require minimisation: one can use the almost linear algorithm by Hopcroft and Karp [33, 1]. This algorithm proceeds as follow: starting from two DFAs $\langle u_1, M_1, v_1\rangle$ and $\langle u_2, M_2, v_2\rangle$, it first computes the disjoint union automaton $\langle u, M, v\rangle$, defined by

$$u = \begin{bmatrix} u_1 & u_2 \end{bmatrix} \qquad M = \begin{bmatrix} M_1 & 0 \\ 0 & M_2 \end{bmatrix} \qquad v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} .$$

It then checks that the former initial states are equivalent by coinduction. Intuitively, two states are equivalent if they can match each other's transitions to reach equivalent states, with the constraint that no accepting state can be equivalent to a non-accepting one.

Let us execute this algorithm on the simple example given on the left-hand side of Fig. 18. We start with the pair of states (x, u); these two states are non-accepting so that we can declare them equivalent *a priori*. We then have to check that they can match each other's transitions, i.e., that y and v are equivalent. Both states are accepting, we declare them equivalent, and we move to the pair (z, w) (according to the transitions of the automata). Again, since these two states are non-accepting, we declare them equivalent and we follow their transitions. This brings us back to the pair (y, v). Since this pair was already encountered, we can stop: the two automata are equivalent, they recognise the same language. The algorithm always terminates: there are finitely many pairs of states, and each pair is visited at most once.

This presentation of the algorithm makes it quadratic in worst case. Almost linear time complexity is obtained by recording a set of equivalence classes rather than the set of visited pairs. To illustrate this idea, consider the example on the right-hand side of Fig. 18: starting from the pair (x, u) and following transitions along a, we reach a situation where the pairs (x, u), (y, v), (z, w), and (z, v) have been declared as equivalent and where we still need to check transitions along b. All of them result in already declared pairs, except the initial one (x, u), which yields (y, w). Although this pair was not visited, it belongs to the equivalence relation generated by the previously visited pairs. Therefore, there is no need to add this pair, and the algorithm can stop immediately. This makes the algorithm almost linear: two equivalence classes are merged at each step of the loop so that this loop is executed at most $n + m$ times, where n and m are the number of states of the compared DFAs. Using a disjoint-sets data structure for maintaining equivalence classes ensures that each step is done in almost-constant time [19].

To our knowledge, there is only one implementation of disjoint-sets in Coq [44]. However, this implementation uses `sig` types to ensure basic invariants along computations, so that reduction of the corresponding terms inside Coq is not optimal: useless proof terms are constantly built and thrown away. Although this drawback disappears when the code is extracted (the goal in [44] was to obtain a certified compiler, by extraction), this is problematic in our case: since we build a reflexive tactic, computations are performed inside Coq. Conchon and Filliâtre also certified a persistent union-find data structure in Coq [17], but this development consists in a modelling of an OCaml library, not in a proper Coq implementation that could be used to perform computations. Therefore, we had to re-implement and prove this data structure from scratch. Namely, we implemented disjoint-sets forests [19] with path compression and the usual "union by rank" heuristic, along the lines of [44], but without using `sig`-types.

We do not give the Coq code for checking equivalence of DFAs here: it closely follows [1] and can be downloaded from [9]. Note that since recursion is not structural, we need to explicitly bound the recursion depth. As explained above, the size of the disjoint union automaton $(n + m)$ does the job.

Like previously, the correctness of this last step reduces to algebraic reasoning. Define a 0-1 matrix Y to encode the equivalence relation on states obtained with a successful run of the algorithm:

$$Y_{ij} = \begin{cases} 1 & \text{if states } i \text{ and } j \text{ are equivalent,} \\ 0 & \text{otherwise.} \end{cases}$$

We prove that this matrix satisfies the following properties (like for the determinisation step, these proofs are quite technical and correspond to a detailed analysis of the algorithm—in particular, we have to show that the bound we impose for the recursion depth is appropriate):

$$1 \leq Y \quad (1) \qquad\qquad Y \cdot Y \leq Y \quad (2) \qquad\qquad Y \cdot M \leq M \cdot Y \quad (3)$$

$$[u_1\ 0] \cdot Y = [0\ u_2] \cdot Y \quad (4) \qquad\qquad Y \cdot v = v \quad (5)$$

Equations $(1, 2)$ correspond to the fact that Y encodes a reflexive and transitive relation. Equation (3) comes from the fact that Y is a simulation: transitions starting from related states yield related states. The last two equations assess that the starting states are related (4), and that related states are either accepting or non-accepting (5).

This allows us to conclude using algebraic reasoning: from $(1, 2, 3)$ and Kleene algebra laws, we deduce

$$M^\star \cdot Y = Y \cdot (M \cdot Y)^\star \ . \tag{6}$$

Also notice that as a special case of (‡), we have

$$M^\star = \begin{bmatrix} M_1 & 0 \\ 0 & M_2 \end{bmatrix}^\star = \begin{bmatrix} M_1^\star & 0 \\ 0 & M_2^\star \end{bmatrix} \ ,$$

so that we have $u_1 \cdot M_1^\star \cdot v_1 = [u_1\ 0] \cdot M^\star \cdot v$ and $u_2 \cdot M_2^\star \cdot v_2 = [0\ u_2] \cdot M^\star \cdot v$. Correctness follows:

$$\begin{aligned} u_1 \cdot M_1^\star \cdot v_1 &= [u_1\ 0] \cdot M^\star \cdot v \\ &= [u_1\ 0] \cdot M^\star \cdot Y \cdot v && \text{(by 5)} \\ &= [u_1\ 0] \cdot Y \cdot (M \cdot Y)^\star \cdot v && \text{(by 6)} \\ &= [0\ u_2] \cdot Y \cdot (M \cdot Y)^\star \cdot v && \text{(by 4)} \\ &= [0\ u_2] \cdot M^\star \cdot Y \cdot v && \text{(by 6)} \\ &= [0\ u_2] \cdot M^\star \cdot v = u_2 \cdot M_2^\star \cdot v_2 \ . && \text{(by 5)} \end{aligned}$$

In other words, we obtained the bottom line equality of Fig. 14.

4.6. **Putting it all together.** By combining the proofs from the above sections according to Fig. 14, we obtain the decision procedure and its correctness proof:

```
Definition regex_to_DFA x :=
  let x' := ssf x in
  let A1 := regex_to_eNFA x' in
  let A2 := eNFA_to_NFA A1 (construction_wf (ssf_complete x)) in
  let A3 := NFA_to_DFA A2 in
    A3.
Definition decide_kleene x y := DFA_equiv (regex_to_DFA x) (regex_to_DFA y).
Theorem decide_kleene_correct: forall x y, decide_kleene x y = true → x == y.
```

As explained in Sect. 2.3, although the above equality lies in the syntactic model of regular expressions, we can actually port it to any model of typed Kleene algebras using reification and the untyping theorem.

4.7. **Completeness: counter-examples.** As announced in Sect. 1.3, we also proved the converse implication, i.e., *completeness*. This basically amounts to exhibiting a counter-example in the case where the DFAs are not equivalent. From the algorithmic point of view, it suffices to record the word that is being read in the algorithm from Sect. 4.5; when two states that should be equivalent differ by their accepting status, we know that the current word is accepted by one DFA and not by the other one. Accordingly, the `decide_kleene` function actually returns an `option (list label)` rather than a Boolean, so that the counter-example can be given to the user—in particular, in the above statement of `decide_kleene_correct`, the constant `true` should be replaced by `None`. We can then get the converse of `decide_kleene_correct`:

```
Theorem decide_kleene_complete: forall x y w, decide_kleene x y = Some w → ¬(x == y).
```

The proof consists in showing that the word `w` possibly returned by the equivalence check algorithm is actually a counter-example, and that the language accepted by a DFA is exactly the language obtained by interpreting the regular expression returned by `DFA.eval`:

```
Definition DFA_language: DFA.t → language := ...
Definition regex_language: regex → language := ...
Lemma language_DFA_eval: forall A, DFA_language A == regex_language (DFA.eval A).
```

(Recall that languages—predicates over lists of letters—form a Kleene algebra which we defined in Fig. 5; in particular, the above symbol `==` denotes equality in this model, i.e., pointwise equivalence of the predicates.) The function `DFA.eval` corresponds to a matricial product (Fig. 13) so that the above lemma requires us to work with matrices over languages. This is actually the only place in the proof where we need this model.

5. Efficiency

Thanks to the efficient reduction mechanism available in Coq [28], and since we carefully avoided mixing proofs with code, the tactic returns instantaneously on typical use cases. We had to perform some additional tests to check that the decision procedure actually scales on larger expressions. This would be important, for example, in a scenario where equations to be solved by the tactic are generated automatically, by an external tool.

A key factor is the concrete representation of numbers, which we detail first.

5.1. Numbers, finite sets, and finite maps.

To code the decision procedure, we mainly needed natural numbers, finite sets, and finite maps. Coq provides several representations for natural numbers: Peano integers (`nat`), binary positive numbers (`positive`), and big natural numbers in base 2^{31} (`BigN.t`), the latter being shipped with an underlying mechanism to use machine integers and perform efficient computations. (On the contrary, unary and binary numbers are allocated on the heap, as any other datatype.) Similarly, there are various implementations of finite maps and finite sets, based on ordered lists (`FMapList`), AVL trees (`FMapAVL`), or uncompressed Patricia trees (`FMapPositive`).

While Coq standard library features well-defined interfaces for finite sets and finite maps, the different definitions of numbers lack this standardisation. In particular, the provided tools vary greatly depending on the implementation. For example, the tactic `omega`, which decides Presburger's arithmetic on `nat`, is not available for `positive`. To abstract from this choice of basic data structures, and to obtain a modular code, we designed a small interface to package natural numbers together with the various operations we need, including sets and maps. We specified these operations with respect to `nat`, and we defined several automation tactics. In particular, by automatically translating goals to the `nat` representation, we can use the `omega` tactic in a transparent way.

We defined several implementations of this interface, so that we could experiment with the possible choices and compare their performances. Of course, unary natural numbers behave badly since they bring an additional exponential factor. However, thanks to the efficient implementation of radix-2 search trees for finite maps and finite sets (`FMapPositive` and `FSetPositive`), we actually get higher performances by using `positive` binary numbers rather than machine integers (`BigN.t`). This is no longer true with the extracted code: using machine integers is faster on large expressions with a thousand internal nodes.

It would be interesting to rework our code to exploit the efficient implementation of persistent arrays in experimental versions of Coq [3]. We could reasonably hope to win an order of magnitude by doing so; this however requires a non-trivial interfacing work since our algorithms were written for dynamically extensible maps over unbounded natural numbers while persistent arrays are of a fixed size, and over cyclic 31 bits integers.

5.2. **Benchmarks.** Two alternative certified decision procedures for regular expression equivalence have been developped since we proposed the present one; both of them rely on a simple algorithm based on Brzozowski's derivatives [13, 51]:

- Krauss and Nipkow [42] implemented a tactic for Isabelle/HOL;
- Coquand and Siles [18] implemented their algorithm in Coq; they use a particularly nice induction scheme for finite sets, which is one of their main contributions.

We performed some benchmarks to compare the performances of these two implementations with ours (we leave the comparison of our approaches for the related works section, Sect. 6.1). The timings are given in Table 1, they have been obtained as follows.

For each pair (n, v) given in the first two columns, we generated 500 pairs of regular expressions, with exactly n nodes and at most v distinct variables[1]. Since two random expressions tend to always be trivially distinct, we artificially modified these pairs to make them equivalent, by adding the full regular expression on both sides. For instance, the pair $(a+b^\star,\ a\cdot b\cdot c)$, with four nodes and three variables, is turned into the pair $(a+b^\star+(a+b+c)^\star,\ a\cdot b\cdot c+(a+b+c)^\star)$. By doing so, we make sure that all algorithms actually explore the whole DFAs corresponding to the initial expressions.

For each of these modified pairs, we measured the time required by each implementation (CoSi, KrNi, and BrPo respectively stand for Coquand and Siles' implementation, Krauss and Nipkow' one, and ours). The timings were measured on a Macbook pro (Intel Core 2 Duo, 2.5GHz, 4Go RAM) running Mac OS X 10.6.7, with Coq 8.3 and Isabelle 2011-1. All times are given in seconds, they correspond to the tactic scenario, where execution takes place inside Coq or Isabelle. (When extracting our Coq procedure to OCaml, the resulting code executes approximately 20 times faster.)

The highly stochastic behaviour of the three algorithms makes this data hard both to compute and to analyse: while the algorithms answer in a reasonably short amount of time for a lot of pairs, there are a few difficult pairs which require a lot of time (up to hours). Therefore, we had to impose timeouts to perform these tests: a "$>$" symbol in Table 1 means that we only have a lower bound for the corresponding cell. Also, since Coquand and Siles' algorithm gives extremely bad performances for medium to large expressions, we could not include timings for this algorithm in the lower rows of this table.

The mean time is reported in the fourth column. Our implementation is an order of magnitude faster than the other ones—even several orders w.r.t. CoSi for non-trivial expressions. However, this mean times are not representative of the actual behaviour of the algorithms: they do not properly account for their behaviour on the few difficult pairs which require a lot of time (both because their weight is low since they are few, and because 500 pairs are not enough to capture difficult pairs in a uniform way). This is why we include the four remaining columns. For each of these columns, say the one entitled "90%", we computed the time which is sufficient to solve at least 90% of the pairs. In other words, the column 50% corresponds to the median times, the column 90% to the last deciles, 99% to the last percentiles, and 100% to the maximal recorded times. For instance, with 20 nodes and 2 variables, 90% of pairs were solved within 0.152 seconds with KrNi; equivalently, 10% pairs required more than 0.152 seconds.

We also report in Fig. 19 the distribution of the timings we obtained for the pairs with 100 nodes and at most 10 variables, with KrNi and BrPo. The time is discretized into 5ms intervals; note the horizontal logarithmic scale, which explains why the area under the

[1]these pairs are available on the web for the interested reader [9].

nodes	vars	algo.	mean	50%	90%	99%	100%
5	2	CoSi	0.017	0.014	0.028	0.070	0.190
		KrNi	0.067	0.066	0.071	0.091	0.097
		BrPo	0.001	0.001	0.002	0.002	0.011
10	2	CoSi	0.070	0.047	0.136	0.516	1.456
		KrNi	0.072	0.070	0.076	0.104	0.131
		BrPo	0.002	0.002	0.002	0.003	0.004
20	2	CoSi	2.037	0.326	2.925	44.101	145.875
		KrNi	0.134	0.135	0.152	0.166	0.534
		BrPo	0.003	0.003	0.004	0.006	0.007
20	4	CoSi	>31.043	13.708	42.983	261.022	>3600.000
		KrNi	0.132	0.122	0.160	0.171	0.685
		BrPo	0.006	0.006	0.008	0.012	0.016
50	4	CoSi	–	–	–	–	–
		KrNi	0.251	0.236	0.294	1.061	2.337
		BrPo	0.019	0.018	0.028	0.049	0.057
100	10	CoSi	–	–	–	–	–
		KrNi	0.686	0.487	0.976	6.401	8.468
		BrPo	0.135	0.128	0.190	0.314	0.359
200	20	CoSi	–	–	–	–	–
		KrNi	>30.420	2.123	16.026	1621.541	>3600.000
		BrPo	0.695	0.662	0.948	1.320	1.672
500	50	CoSi	–	–	–	–	–
		KrNi	>280.340	96.220	>900.000	>3600.000	>4703.633
		BrPo	6.007	5.676	8.103	10.912	11.949
1000	100	CoSi	–	–	–	–	–
		KrNi	–	–	–	–	–
		BrPo	29.651	27.393	41.917	59.072	70.150

Table 1. Benchmarks for the existing certified decision procedures.

KrNi curve looks smaller than the area under the BrPo curve (while they are both equal to 500, the number of checked pairs). This kind of expressions (100 nodes, 10 variables) corresponds to the line in Table 1 where the two algorithms are the closest in terms of performances; we can however notice that while the median values are comparable, KrNi suffers from a rather long trail: there is a difference of one order of magnitude for the last percentile.

For larger expressions (500 to 1000 nodes), our tactic clearly outperforms the two other ones, in terms of both mean time, median time, and worst cases trail. In particular, our implementaion seems to be much more robust w.r.t. difficult pairs: in Table 1, the value of the last percentile is always roughly equal to twice the median value, so that the mean is always almost equal to the median.

The particular care we took to implement all steps of our procedure in an efficient way could partially explain the observed performance gap; however, our intuition is that this gap mainly comes from the construction algorithm we use (by Ilie and Yu [34]), which produces much smaller automata than the ones obtained with Brzozowski derivatives [13].

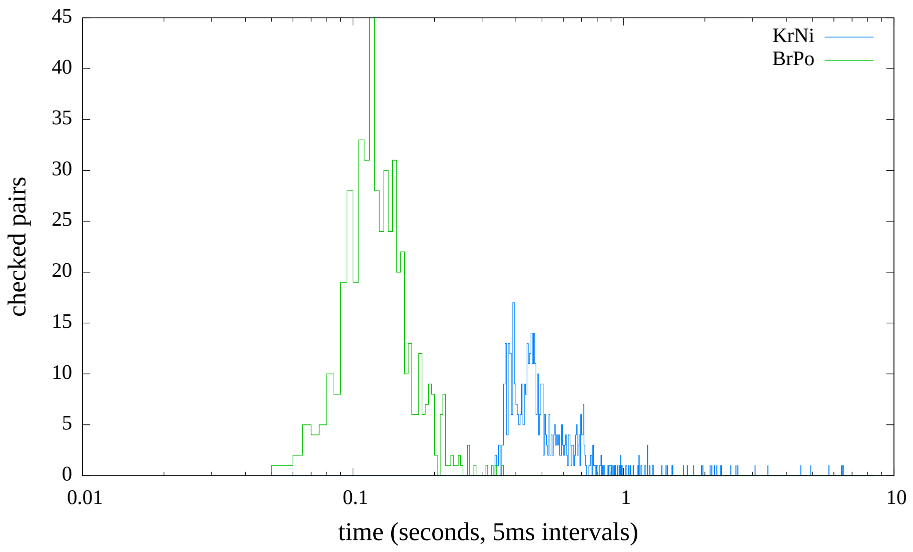


Figure 19. Distribution of the timings measured with Krauss and Nipkow' algorithm and ours (for the 500 pairs with 100 nodes and at most 10 variables from Table 1).

6. Conclusions

We presented a correct and complete reflexive tactic for deciding Kleene algebra equalities. This tactic belongs to a broader project whose aim is to provide algebraic tools for working with binary relations in Coq. The development is axiom-free, it can be downloaded from [9]. To our knowledge, this is the first certified efficient implementation of these algorithms and their integration as a generic tactic.

According to `coqwc`, the development consists of approximately 10.000 lines of Coq code, which distribute as follows and to which we must add a 350 lines OCaml file for performing reification:

	specifications	proofs	comments
infrastructure	1959	1139	486
models	797	313	98
matrices	633	510	93
decision procedure	1716	2353	261
total	5105	4315	938

The *infrastructure* line corresponds to the basic infrastructure files, the definition of the algebraic hierarchy using typeclasses, and basic lemmas and tactics for monoids, semi-lattices, idempotent semirings, and Kleene algebras. As expected, this part is rather verbose. The *models* line is for the definition of the various models, including languages, binary relations, and regular expressions; proofs are either trivial or fully automatised in this part. The *matrices* line corresponds to all matrix constructions (up to the fact that matrices form a Kleene algebra); proofs are eased by the tactics we defined in the infrastructure but they are not fully automatic: they follow standard paper proofs. The remaining line corresponds to the decision procedure itself. As expected, this is where the ratio proofs/specification is the largest: although we exploit high-level tactics to perform case analyses, or `omega` to reason about arithmetic, most proofs are non-trivial and have to be rather explicit.

6.1. **Related works.**

Algebraic tools for binary relations. The idea of reasoning about binary relations algebraically is old [55, 22]. Among others [36, 57], Struth applied this idea within an interactive theorem prover [54]. He later turned to automated first-order theorem provers (ATP): Höfner and him verified facts about various relation algebras [31, 32] using Prover9, a resolution/paramodulation based ATP. Our approaches are quite different: we implemented a decision procedure for a decidable theory, whereas their proposal consists in feeding a generic automated prover with the axioms of some algebras, and to see how far the prover can go by itself. As a consequence, their methodology applies directly to a very wide class of goals and algebras, while we are restricted to the equational theory of Kleene algebras. On the other hand, our tactic always terminates, while Prover9 is unpredictable: even for very simple goals, it can diverge, find a proof immediately, or find a proof in a few minutes [32]. Foster, Struth, and Weber recently used Isabelle/HOL to formalise proofs about relation algebras [23]. While our long-term goals are very close, our approaches and results are quite different, for the same reasons as above: we focused on a single tactic to solve the whole equational theory of Kleene algebra, while they use generic automatic methods that are applicable to a much wider class of goals, at the cost of requiring user-guidance if the goal is not simple enough.

Narboux defined a set of Coq tactics for diagrammatic proofs [47]. He works in the concrete setting of binary relations, which makes it possible to represent more diagrams, but does not scale to other models. The level of automation is rather low: it basically reduces to a set of hints for the `auto` tactic.

Finite automata theory. The notion of strict star form (Sect. 4.3) was inspired by the standard notion of *star normal form* [12] and the idea of *star unavoidability* [34]. To our knowledge, using this notion to get ϵ-NFAs with acyclic epsilon-transitions is a new idea.

At the time we started this project, Briais formalised decidability of regular languages equality [11] (but not Kozen's initiality theorem). However, his approach is not computational, so that even straightforward identities cannot be checked by letting Coq compute.

The Isabelle/HOL tactic implemented by Nipkow and Krauss to decide regular expressions equivalence [42] is simpler than the one we presented here, for several reasons. First, they implemented an algorithm based on Brzozowski's derivatives [13, 51], which is less involved than ours, but also less efficient: the DFAs are produced directly from the regular expressions, but they can be much larger [34]. This certainly explains the performance gaps we observed in Sect. 5.2. Second, they do not prove Kozen's initiality theorem: they prove correctness in the model of regular languages and they use a nice mathematical trick to reach the model of binary relations. As a consequence, their tactic cannot be used with other models like matrices, $(min, +)$ algebras, or weighted relations (graphs whose vertices are labelled by the elements of an arbitrary Kleene algebra). Third, they do not formalise the proof of completeness, or equivalently, the fact that the algorithm always terminates (Isabelle/HOL computations do not need to terminate so that they can use a "while-option" combinator). For all these reasons, their development is much more concise than ours.

Coquand and Siles' recent implementation of the same algorithm than Krauss and Nipkow in Coq [18] is not efficient, and cannot reliably be used for expressions with more than twenty nodes (see Table 1). A possible explanation could be that they mix proofs and computations: this is known to be problematic since proofs then have to be passed around along reductions, even with `vm_compute`—the efficient Coq normalisation function [28]. Like Krauss and Nipkow, they do not formalise Kozen's initiality theorem; they prove the completeness of their algorithm, though.

Formalisation of algebraic hierarchies. The problem of formalising mathematical structures or algebraic hierarchies in type theory is well-known and usually considered as difficult [4, 6, 26, 14, 25]. Thanks to the recent addition of first-class typeclasses [52], we can use a very simple and naive solution here, which gives us overloading for notations, lemmas, and tactics, as well as modularity, sharing, and a basis for reification (Sect. 2).

Since we started this project, Spitters and van der Weegen also described how to use typeclasses to define an algebraic hierarchy [53]. Leaving apart the fact that we work with typed structures, they follow the strategy we presented here (and previously in [10]); in particular, they use separate classes for operations and laws, and they attach notations to class projections. They actually use an even stronger discipline: each operation comes with a class (e.g., our `Monoid_Ops` class corresponds to their classes `SemiGroupOp` and `MonoidUnit`).

We discussed two drawbacks of this approach in Sect. 2.4, the most important one from our point of view being the difficulty we had when trying to work with richer structures. Indeed, the hierarchy we need for this work is really small (it has depth three where the one from [25] had depth ten at the time of writing), so that there are few instances to declare for typeclass resolution. As a consequence, typeclass resolution is efficient and the approach works out of the box. On the contrary, our attempts to define richer structures were rather frustrating. There are many more instances to declare (these include all the inheritance relationships, all model constructions like matrices, all the compatibility lemmas that give the ability to rewrite using user-defined relations). Thus, typeclass resolution becomes too slow to be used in practice—when we manage not to introduce infinite loops, which also happens to be difficult.

Therefore, for rather large algebraic hierarchies, it is unclear to us whether one should pursue with this simple approach, betting that these problems can be resolved by improving the implementation of typeclasses. Despite their apparent complexity, solutions like the ones proposed in [25] might be less hazardous.

6.2. **Directions for future work.** We conclude with possible directions for future work.

Earlier failure checks. Our algorithm for checking equivalence of DFAs returns whenever two non-equivalent states are encountered. This makes the tactic faster in case of failure, which is interesting when the tactic is used in a "try" block, where failures are expected to happen. We could actually go one step further, by checking the equivalence on-the-fly, during the determinisation phase. This means computing the DFAs lazily and stopping as soon as a discrepancy is found. Doing so, we would avoid the potentially expensive computation of the whole DFAs in case of failure. Although this approach is definitely more efficient than the current one for the case of failures, it introduces some difficulties in the correctness proof, which we did not complete.

A simpler proof of initiality. Since we wanted to get a tactic for all models of Kleene algebras, we had to formalise Kozen's initiality proof. With this goal in mind, the derivative-based algorithm implemented by Nipkow and Krauss [42] is quite appealing for its simplicity. Moreover, since the notion of derivative is purely syntactic, it is very well suited to algebraic reasoning. However, rather surprisingly, we could not find a way to replay Kozen's initiality proof with this algorithm. We leave this question for future work.

KAT, Hoare logic. We plan to extend our decision procedure to deal with *Kleene algebras with tests* (KAT), so as to provide automation to prove correctness of programs in Hoare logic [40]. A first possibility would be to encode KAT expressions into KA [41] and to use the current tactic. This encoding being exponential in the number of predicate variables, it is unclear whether this approach would be tractable. A more involved approach would be to use the dedicated automata construction presented in [16].

Richer algebras. Kleene algebras lack several important operations from binary relations: intersection, converse, complement, residuals... We plan to develop other tools for algebras dealing with these operators, like *Kleene algebras with converse* [20], *residuated Kleene lattices* [35], or *allegories* [24]. In particular, residuated structures provide means of encoding properties like well-foundedness [22], which are quite important for program semantics. These structures are not known to be decidable; waiting for new algorithms to be found, we can already build on our library to implement various tools for working with these structures in the Coq proof assistant.

Acknowledgements. We warmly thank Guilhem Moulin, Assia Mahboubi, Matthieu Sozeau, Bruno Barras, and Hugo Herbelin for highly stimulating discussions. We are also grateful to the anonymous referees of the first Coq workshop in 2009 and ITP in 2010, whose remarks helped us to improve both the library and its description.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] S. F. Allen, R. L. Constable, D. J. Howe, and W. E. Aitken. The semantics of reflected proof. In *LICS*, pages 95–105. IEEE Computer Society, 1990.

[3] M. Armand, B. Grégoire, A. Spiwack, and L. Théry. Extending Coq with imperative features and its application to SAT verification. In *Proc. ITP*, volume 6172 of *Lecture Notes in Computer Science*, pages 83–98. Springer Verlag, 2010.

[4] G. Barthe. Implicit coercions in type systems. In *TYPES*, volume 1158 of *Lecture Notes in Computer Science*, pages 1–15. Springer Verlag, 1995.

[5] Y. Bertot, G. Gonthier, S. Ould Biha, and I. Pasca. Canonical big operators. In *Proc. TPHOL*, volume 5170 of *Lecture Notes in Computer Science*, pages 86–101. Springer Verlag, 2008.

[6] G. Betarte and A. Tasistro. Formalisation of systems of algebras using dependent record types and subtyping: an example. In *Proc. 7th Nordic workshop on Pro- gramming Theory*, 1995.

[7] F. Blanqui, S. Coupet-Grimal, W. Delobel, and A. Koprowski. CoLoR: a Coq library on rewriting and termination, 2006.

[8] R. Boyer and J. Moore. Metafunctions: proving them correct and using them efficiently as new proof procedures. In *The Correctness Problem in Computer Science*. NY: Academic Press, 1981.

[9] T. Braibant and D. Pous. Coq library: ATBR, algebraic tools for working with binary relations. `http://sardes.inrialpes.fr/~braibant/atbr/`, May 2009.

[10] T. Braibant and D. Pous. An efficient Coq tactic for deciding Kleene algebras. In *Proc. ITP*, volume 6172 of *Lecture Notes in Computer Science*, pages 163–178. Springer Verlag, 2010.

[11] S. Briais. Coq development: Finite automata theory. `http://www.prism.uvsq.fr/~bris/tools/Automata_080708.tar.gz`, July 2008.

[12] A. Brüggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120(2):197–213, 1993.

[13] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

[14] C. Sacerdoti Coen and E. Tassi. Working with mathematical structures in Type Theory. In *Proc. TYPES*, volume 4941 of *Lecture Notes in Computer Science*, pages 157–172. Springer Verlag, 2007.

[15] C. Sacerdoti Coen and E. Tassi. Nonuniform coercions via unification hints. In *Proc. TYPES*, volume 53 of *EPTCS*, pages 16–29, 2009.

[16] E. Cohen, D. Kozen, and F. Smith. The complexity of Kleene algebra with tests, 1996. TR96-1598, CS Dpt., Cornell University.

[17] S. Conchon and J.-C. Filliâtre. A Persistent Union-Find Data Structure. In *Proc. ACM SIGPLAN Workshop on ML*, pages 37–45, Freiburg, Germany, October 2007.

[18] T. Coquand and V. Siles. A decision procedure for regular expression equivalence in type theory. In *Proc. CPP*, volume 7086 of *Lecture Notes in Computer Science*, pages 119–134. Springer Verlag, 2011.

[19] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, second edition, 2001.

[20] S. Crvenkovic, I. Dolinka, and Z. Ésik. The variety of Kleene algebras with conversion is not finitely based. *Theoretical Computer Science*, 230(1-2):235–245, 2000.

[21] Brian Davey and Hilary Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[22] H. Doornbos, R. Backhouse, and J. van der Woude. A calculational approach to mathematical induction. *Theoretical Computer Science*, 179(1-2):103–135, 1997.

[23] S. Foster, G. Struth, and T. Weber. Automated engineering of relational and algebraic methods in Isabelle/HOL - (invited tutorial). In *Proc. RAMICS*, volume 6663 of *Lecture Notes in Computer Science*, pages 52–67. Springer Verlag, 2011.

[24] P. Freyd and A. Scedrov. *Categories, Allegories*. North Holland, 1990.

[25] F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging mathematical structures. In *Proc. TPHOL*, volume 5674 of *Lecture Notes in Computer Science*, pages 327–342. Springer Verlag, 2009.

[26] H. Geuvers, R. Pollack, F. Wiedijk, and J. Zwanenburg. A constructive algebraic hierarchy in Coq. *Journal of Symbolic Computation*, 34(4):271–286, 2002.

[27] G. Gonthier. Point-free, set-free concrete linear algebra. In *Proc. ITP*, volume 6898 of *Lecture Notes in Computer Science*, pages 103–118. Springer Verlag, 2011.

[28] B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *Proc. ICFP*, pages 235–246, 2002.

[29] B. Grégoire and A. Mahboubi. Proving equalities in a commutative ring done right in Coq. In *TPHOL*, volume 3603 of *Lecture Notes in Computer Science*, pages 98–113. Springer Verlag, 2005.

[30] M. Hedberg. A Coherence Theorem for Martin-Löf's Type Theory. *J. of Functional Programming*, 8(4):413–436, 1998.

[31] P. Höfner and G. Struth. Automated reasoning in Kleene algebra. In *Proc. CADE*, volume 4603 of *Lecture Notes in Computer Science*, pages 279–294. Springer Verlag, 2007.

[32] P. Höfner and G. Struth. On automating the calculus of relations. In *IJCAR*, volume 5195 of *Lecture Notes in Computer Science*, pages 50–66. Springer Verlag, 2008.

[33] J. E. Hopcroft and R. M. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report 114, Cornell University, December 1971.

[34] L. Ilie and S. Yu. Follow automata. *Information and Computation*, 186(1):140–162, 2003.

[35] P. Jipsen. From semirings to residuated Kleene lattices. *Studia Logica*, 76(2):291–303, 2004.

[36] W. Kahl. Calculational relation-algebraic proofs in Isabelle/Isar. In *Proc. RelMiCS*, volume 3051 of *Lecture Notes in Computer Science*, pages 178–190. Springer Verlag, 2003.

[37] S. C. Kleene. Representation of events in nerve nets and finite automata. In *Automata Studies*, pages 3–41. Princeton University Press, 1956.

[38] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, 1994.

[39] D. Kozen. Typed Kleene algebra, 1998. TR98-1669, CS Dpt. Cornell University.

[40] D. Kozen. On Hoare logic and Kleene algebra with tests. *ACM Trans. Comput. Log.*, 1(1):60–76, 2000.

[41] D. Kozen and F. Smith. Kleene algebra with tests: Completeness and decidability. In *CSL*, volume 1258 of *Lecture Notes in Computer Science*, pages 244–259. Springer Verlag, September 1996.

[42] A. Krauss and T. Nipkow. Proof pearl: Regular expression equivalence and relation algebra, 2011. To appear in J. of Algebraic Reasoning.

[43] D. Krob. Complete systems of B-rational identities. *Theoretical Computer Science*, 89(2):207–343, 1991.

[44] X. Leroy. A formally verified compiler back-end. *J. of Algebraic Reasoning*, 43(4):363–446, 2009.

[45] Saunders Mac Lane. *Categories for the Working Mathematician.* Number 5 in Graduate Texts in Mathematics. Springer Verlag, 2nd edition, 1998.

[46] A.R. Meyer and L. J. Stockmeyer. Word problems requiring exponential time. In *Proc. STOC*, pages 1–9. ACM, 1973.

[47] J. Narboux. *Formalisation et automatisation du raisonnement géométrique en Coq.* PhD thesis, Université Paris Sud, September 2006.

[48] D. Pous. Untyping typed algebraic structures and colouring proof nets of cyclic linear logic. In *Proc. CSL*, volume 6247 of *Lecture Notes in Computer Science*, pages 484–498. Springer Verlag, August 2010.

[49] M.O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959.

[50] V. Redko. On defining relations for the algebra of regular events. *Ukrain. Mat. Z.*, 16:120–126, 1964.

[51] J. J. M. M. Rutten. Automata and coinduction (an exercise in coalgebra). In *Proc. CONCUR*, volume 1466 of *Lecture Notes in Computer Science*, pages 194–218. Springer Verlag, 1998.

[52] M. Sozeau and N. Oury. First-class type classes. In *Proc. TPHOL*, volume 4732 of *Lecture Notes in Computer Science*, pages 278–293. Springer Verlag, 2008.

[53] B. Spitters and E. van der Weegen. Type classes for mathematics in type theory. *MSCS, special issue on 'Interactive theorem proving and the formalization of mathematics'*, 21:1–31, 2011. (A four pages abstract appeared in Proc. ITP 2010.).

[54] G. Struth. Calculating Church-Rosser proofs in Kleene algebra. In *Proc. RelMiCS*, volume 2561 of *Lecture Notes in Computer Science*, pages 276–290. Springer Verlag, 2001.

[55] A. Tarski and S. Givant. *A Formalization of Set Theory without Variables*, volume 41 of *Colloquium Publications.* American Mathematical Society, Providence, Rhode Island, 1987.

[56] K. Thompson. Regular expression search algorithm. *C. ACM*, 11:419–422, 1968.

[57] D. von Oheimb and T.F. Gritzner. RALL: Machine-supported proofs for relation algebra. In *Proc. CADE*, volume 1249 of *Lecture Notes in Computer Science*, pages 380–394. Springer Verlag, 1997.

Tactics for Reasoning modulo AC in Coq[⋆]

Thomas Braibant and Damien Pous[⋆⋆]

LIG, UMR 5217, CNRS, INRIA, Grenoble

Abstract. We present a set of tools for rewriting modulo associativity and commutativity (AC) in Coq, solving a long-standing practical problem. We use two building blocks: first, an extensible reflexive decision procedure for equality modulo AC; second, an OCaml plug-in for pattern matching modulo AC. We handle associative only operations, neutral elements, uninterpreted function symbols, and user-defined equivalence relations. By relying on type-classes for the reification phase, we can infer these properties automatically, so that end-users do not need to specify which operation is A or AC, or which constant is a neutral element.

1 Introduction

Motivations. Typical hand-written mathematical proofs deal with commutativity and associativity of operations in a liberal way. Unfortunately, a proof assistant requires a formal justification of all reasoning steps, so that the user often needs to make boring term re-orderings before applying a theorem or using a hypothesis. Suppose for example that one wants to rewrite using a simple hypothesis like `H`: ∀`x`, `x`+−`x` = 0 in a term like `a`+`b`+`c`+−(`c`+`a`). Since Coq standard `rewrite` tactic matches terms syntactically, this is not possible directly. Instead, one has to reshape the goal using the commutativity and associativity lemmas:

```
                                                (* ⊢ ((a+b)+c)+-(c+a) = ... *)
rewrite (add_comm a b), ← (add_assoc b a c).    (* ⊢ (b+(a+c))+-(c+a) = ... *)
rewrite (add_comm c a), ← add_assoc.            (* ⊢ b+((a+c)+-(a+c)) = ... *)
rewrite H.                                      (* ⊢ b+0              = ... *)
```

This is not satisfactory for several reasons. First, the proof script is too verbose for such a simple reasoning step. Second, while reading such a proof script is easy, writing it can be painful: there are several sequences of rewrites yielding to the desired term, and finding a reasonably short one is difficult. Third, we need to copy-paste parts of the goal to select which occurrence to rewrite using the associativity or commutativity lemmas; this is not a good practice since the resulting script breaks when the goal is subject to small modifications. (Note that one could also select occurrences by their positions, but this is at least as difficult for the user which then has to count the number of occurrences to skip, and even more fragile since these numbers cannot be used to understand the proof when the script breaks after some modification of the goal.)

In this paper, we propose a solution to this short-coming for the Coq proof-assistant: we extend the usual rewriting tactic to automatically exploit associativity and commutativity (AC), or just associativity (A) of some operations.

[⋆] Appeared in Proc. CPP, volume 7086 of LNCS, Springer-Verlag, 2011.

[⋆⋆] Supported by "Choco", ANR-07-BLAN-0324 and "PiCoq", ANR-10-BLAN-0305.

Trusted unification vs untrusted matching. There are two main approaches to implementing rewriting modulo AC in a proof-assistant. First, one can extend the unification mechanism of the system to work modulo AC [20]. This option is quite powerful, since most existing tactics would then work modulo AC. It however requires non-trivial modifications of the kernel of the proof assistant (e.g., unification modulo AC does not always yield finite complete sets of unifiers). As a consequence, this obfuscates the meta-theory: we need a new proof of strong normalisation and we increase the trusted code base. On the contrary, we can restrict ourselves to pattern matching modulo AC and use the core-system itself to validate all rewriting steps [8]. We chose this option.

Contributions, scope of the library. Besides the facts that such tools did not exist in Coq before and that they apparently no longer exist in Isabelle/HOL (see §6.1 for a more thorough discussion), the main contributions of this work lie in the way standard algorithms and ideas are combined together to get tactics that are efficient, easy to use, and covering a large range of situations:

– We can have any number of associative and possibly commutative operations, each possibly having a neutral element. For instance, we can have the operations `min`, `max`, $+$, and $*$ on natural numbers, where `max` and $+$ share the neutral element 0, $*$ has neutral element 1, and `min` has no neutral element.
– We deal with arbitrary user-defined equivalence relations. This is important for rational numbers or propositions, for example, where addition and subtraction (resp. conjunction and disjunction) are not AC for Leibniz equality, but for rational equality, `Qeq` (resp. propositional equivalence, `iff`).
– We handle "uninterpreted" function symbols: n-ary functions for which the only assumption is that they preserve the appropriate equivalence relation—they are sometimes called "proper morphisms". For example, subtraction on rational numbers is a proper morphism for `Qeq`, while pointwise addition of numerators and denominators is not. (Note that any function is a proper morphism for Leibniz equality.)
– The interface we provide is straightforward to use: it suffices to declare instances of the appropriate type-classes [22] for the operations of interest, and our tactics will exploit this information automatically. Since the type-class implementation is first-class, this gives the ability to work with polymorphic operations in a transparent way (e.g., concatenation of lists is declared as associative once and for all.)

Methodology. Recalling the example from the beginning, an alternative to explicit sequences of rewrites consists in making a transitivity step through a term that matches the hypothesis' left-hand side syntactically:

```
                                 (* ⊢ ((a+b)+c)+-(c+a) = ... *)
transitivity (b+((a+c)+−(a+c))).
  ring. (* aac_reflexivity *)    (*   ⊢ ((a+b)+c)+-(c+a) = b+((a+c)+-(a+c)) *)
rewrite H.                       (* ⊢ b+((a+c)+-(a+c)) = ... *)
                                 (* ⊢ b+0 = ... *)
```

Although the `ring` tactic [14] solves the first sub-goal here, this is not always the case (e.g., there are AC operations that are not part of a ring structure). Therefore, we have to build a new tactic for equality modulo A/AC: `aac_reflexivity`.

Another drawback is that we also have to copy-paste and modify the term manually, so that the script can break if the goal evolves. This can be a good practice in some cases: the transitivity step can be considered as a robust and readable documentation point; in other situations we want this step to be inferred by the system, by pattern matching modulo A/AC [15].

All in all, we proceed as follows to define our `aac_rewrite` rewriting tactic. Let $\equiv_{AC}$ denote equality modulo A/AC; to rewrite using a universally quantified hypothesis of the form $H : \forall \tilde{x}, p\tilde{x} = q\tilde{x}$ in a goal G, we take the following steps, which correspond to building the proof-tree on the right-hand side:

1. choose a context C and a substitution σ such that $G \equiv_{AC} C[p\sigma]$ (pattern matching modulo AC);
2. make a transitivity step through $C[p\sigma]$;
3. close this step using a dedicated decision procedure (`aac_reflexivity`);
4. use the standard `rewrite`;
5. let the user continue the proof.

$$\dfrac{\dfrac{}{G \equiv_{AC} C[p\sigma]}\;3 \qquad \dfrac{H \qquad \dfrac{\vdots}{C[q\sigma]}\;5}{C[p\sigma]}\;4}{G}\;2$$

For the sake of efficiency, we implement the first step as an OCaml oracle, and we check the results of this (untrusted) matching function in the third step, using the certified decision procedure `aac_reflexivity`. To implement this tactic, we use the standard methodology of *reflection* [8,1,14]. Concretely, this means that we implement the decision procedure as a Coq function over "reified" terms, which we prove correct inside the proof assistant. This step was actually quite challenging: to our knowledge, `aac_reflexivity` is the first reflexive Coq tactic that handles uninterpreted function symbols. In addition to the non-trivial reification process, a particular difficulty comes from the (arbitrary) arity of these symbols. To overcome this problem in an elegant way, our solution relies on a dependently typed syntax for reified terms.

Outline. We sketch the user interface (§2) before describing the decision procedure (§3) and the algorithm for pattern matching modulo AC (§4). We detail our handling of neutral elements and subterms separately (§5). We conclude with related works and directions for future work (§6).

2 User interface, notation

Declaring A/AC operations. We rely on type-classes [22] to declare the properties of functions and A/AC binary operations. This allows the user to extend both the decision procedure and the matching algorithm with new A/AC operations or units in a very natural way. Moreover, this is the basis of our reification mechanism (see §3.2).

The classes corresponding to the various properties that can be declared are given in Fig. 1: being associative, commutative, and having a neutral element. Basically, a user only needs to provide instances of these classes in order to

```
Variables (X: Type) (R: relation X) (op: X → X → X).
Class Associative := law_assoc: ∀x y z, R (op x (op y z)) (op (op x y) z).
Class Commutative := law_comm: ∀x y, R (op x y) (op y x).
Class Unit (e: X) := { law_id_left: ∀x, R (op e x) x; law_id_right: ∀x, R (op x e) x }.

Instance plus_A: Associative eq plus.       Instance and_A: Associative iff and.
Instance plus_C: Commutative eq plus.       Instance and_C: Commutative iff and.
Instance plus_U: Unit eq plus O.            Instance and_U: Unit iff and True.
                                            Instance and_P: Proper (iff ⇒iff ⇒iff) and.
Instance app_A X: Associative eq (app X).   Instance not_P: Proper (iff ⇒iff) not.
Instance app_U X: Unit eq (app X) (nil X).
```

Fig. 1. Classes for declaring properties of operations, example instances.

use our tactics in a setting with new A or AC operations. These classes are parameterised by a relation (`R`): one can use an arbitrary equivalence relation.

Fig. 1 also contains examples of instances. Polymorphic values (`app`, `nil`) are declared in a straightforward way. For propositional connectives (`and`, `not`), we also need to show that they preserve equivalence of propositions (`iff`), since this is not Leibniz equality; we use for that the standard `Proper` type-class—when the relation `R` is Leibniz equality, these instances are inferred automatically. Of course, while we provide these instances, more can be defined by the user.

Example usage. The main tactics we provide are `aac_rewrite`, to rewrite modulo A/AC, and `aac_reflexivity` to decide an equality modulo A/AC. Here is a simple example where we use both of them:

```
H1: ∀x y z, x∩y ∪ x∩z = x∩(y∪z)          Proof.
H2: ∀x y, x∩x = x                          aac_rewrite H1; (* c ∩ (a ∪ b∩d) ∩ c = ... *)
a, b, c, d: set                            aac_rewrite H2; (* c ∩ (a ∪ b∩d)     = ... *)
=====================                      aac_reflexivity.
(a∩c ∪ b∩c∩d) ∩ c = (a ∪ d∩b) ∩ c         Qed.
```

As expected, we provide variations to rewrite using the hypothesis from right to left, or in the right-hand side of the goal.

Listing instances. There might be several ways of rewriting a given equation: several subterms may match, so that the user might need to select which occurrences to rewrite. The situation can be even worse when rewriting modulo AC: unlike with syntactical matching, there might be several ways of instantiating the pattern so that it matches a given occurrence. (E.g., matching the pattern $x + y + y$ at the root of the term $a + a + b + b$ yields two substitutions: $\{x \mapsto a + a; y \mapsto b\}$ and the symmetrical one—assuming there is no neutral element.) To help the user, we provide an additional tactic, `aac_instances`, to display the possible occurrences together with the corresponding instantiations. The user can then use the tactic `aac_rewrite` with the appropriate options.

Notation and terminology. We assume a signature Σ and we let $f, g, h, \ldots$ range over function symbols, reserving letters $a, b, c, \ldots$ for constants (function symbols of arity 0). We denote the set of *terms* by $T(\Sigma)$. Given a set V of variables, we let $x, y, z, \ldots$ range over (universally quantified) variables; a *pattern* is a term with variables, i.e., an element of $T(\Sigma + V)$. A *substitution* (σ) is a partial function that maps variables to terms, which we extend into a partial function from patterns to terms, as expected. Binary function symbols (written with an infix symbol, $\diamond$) can be associative (axiom A) and optionally commutative (axiom C); these symbols may be equipped with a left and right unit u (axiom $U_{u,\diamond}$):

$$A_\diamond : x \diamond (y \diamond z) \equiv (x \diamond y) \diamond z \quad C_\diamond : x \diamond y \equiv y \diamond x \quad U_{u,\diamond} : x \diamond u \equiv x \wedge u \diamond x \equiv x$$

We use $+_i$ (or $+$) for associative-commutative symbols (AC), and $*_i$ (or $*$) for associative only symbols (A). We denote by $\equiv_{AC}$ the equational theory generated by these axioms on $T(\Sigma)$. For instance, in a non-commutative semi-ring $(+, *, 0, 1)$, $\equiv_{AC}$ is generated by A_+, C_+, A_* and $U_{1,*}, U_{0,+}$.

3 Deciding equality modulo AC

In this section, we describe the `aac_reflexivity` tactic, which decides equality modulo AC, is extensible through the definition of new type-class instances, and deals with uninterpreted function symbols of arbitrary arity. For the sake of clarity, we defer the case where binary operations have units to §5.1.

3.1 The algorithm and its proof

A two-level approach. We use the so called 2-level approach [4]: we define an inductive type `T` for terms and a function `eval: T → X` that maps reified terms to user-level terms living in some type `X` equipped with an equivalence relation `R`, which we sometimes denote by $\equiv$. This allows us to reason and compute on the syntactic representation of terms, whatever the user-level model.

We follow the usual practice which consists in reducing equational reasoning to the computation and comparison of normal forms: it then suffices to prove that the normalisation function is correct to get a sound decision procedure.

```
Definition norm: T → T := ...
Lemma eval_norm: ∀u, eval (norm u) ≡ eval u.
Theorem decide: ∀u v, compare (norm u) (norm v) = Eq → eval u ≡ eval v.
```

This is what is called the *autarkic way*: the verification is performed inside the proof-assistant, using the conversion rule. To prove `eval u` $\equiv$ `eval v`, it suffices to apply the theorem `decide` and to let the proof-assistant check by computation that the premise holds. The algorithm needs to meet two objectives. First, the normalisation function (`norm`) must be efficient, and this dictates some choices for the representation of terms. Second, the evaluation function (`eval`) must be simple (in order to keep the proofs tractable) and total: ill-formed terms shall be rejected syntactically.

Packaging the reification environment. We need Coq types to package information about binary operations and uninterpreted function symbols. They are given in Fig. 2, where `respectful` is the definition from Coq standard library for declaring proper morphisms. We first define functions to express the fact that

```
(* type of n-ary homogeneous functions *)
Fixpoint type_of (X: Type) (n: nat): Type :=
  match n with O ⇒ X | S n ⇒ X → type_of X n end.

(* relation to be preserved by n-ary functions *)
Fixpoint rel_of (X: Type) (R: relation X) (n: nat): relation (type_of X n) :=
  match n with O ⇒ R | S n ⇒ respectful R (rel_of n) end.
```

```
Module Bin.
 Record pack X R := {
  value: X → X → X;
  compat: Proper (R ⇒R ⇒R) value;
  assoc: Associative R value;
  comm: option (Commutative R value) }.
End Bin.
```

```
Module Sym.
 Record pack X R := {
  ar: nat;
  value: type_of X ar;
  compat: Proper (rel_of X R ar) value }.

End Sym.
```

Fig. 2. Types for symbols.

n-ary functions are proper morphisms. A "binary package" contains a binary operation together with the proofs that it is a proper morphism, associative, and possibly commutative (we use the type-classes from Fig. 1). An "uninterpreted symbol package" contains the arity of the symbol, the corresponding function, and the proof that this is a proper morphism.

The fact that symbols arity is stored in the package is crucial: by doing so, we can use standard finite maps to store all function symbols, irrespective of their arity. More precisely, we use two environments, one for uninterpreted symbols and one for binary operations; both of them are represented as non-dependent functions from a set of indices to the corresponding package types:

```
Variables (X: Type) (R: relation X).
Variable e_sym: idx → Sym.pack X R.
Variable e_bin: idx → Bin.pack X R.
```

(The type `idx` is an alias for `positive`, the set of binary positive numbers; this allows us to define the above functions efficiently, using binary trees).

Syntax of reified terms. We now turn to the concrete representation of terms. The first difficulty is to choose an appropriate representation for AC and A nodes, to avoid manipulating binary trees. As it is usually done, we flatten these binary nodes using variadic nodes. Since binary operations do not necessarily come with a neutral element, we use non-empty lists (resp. non-empty multisets) to reflect the fact that A operations (resp. AC operations) must have at least one argument. (We could even require A/AC operations to have at least two arguments but this would slightly obfuscate the code and prevent some sharing for multi-sets.) The second difficulty is to prevent ill-formed terms, like "`log` 1 2 3", where a unary function is applied to more than one argument. One could define a predicate stating that terms are well-formed [11], and use this extra hypothesis in later reasonings. We found it nicer to use dependent types to enforce the constraint that symbols are applied to the right number of arguments: it suffices to use vectors of arguments rather than lists. The resulting data-type for reified terms is given in Fig. 3; it depends on the environment for

```
(* non-empty lists/multisets *)
Inductive nelist A :=
| nil: A → nelist A
| cons: A → nelist A → nelist A.

Definition nemset A :=
  nelist (A∗positive).

(* reified terms *)
Inductive T: Type :=
| bin_ac: idx → nemset T → T
| bin_a : idx → nelist T → T
| sym: ∀i, vect T (Sym.ar (e_sym i)) → T.
```

```
Fixpoint eval (u: T): X :=
match u with
| bin_ac i l ⇒ let o:=Bin.value (e_bin i) in
   nefold_map o (fun(u,n)⇒copy o n (eval u)) l
| bin_a i l ⇒ let o:=Bin.value (e_bin i) in
   nefold_map o eval l
| sym i v ⇒ xeval v (Sym.value (e_sym i))
end
with xeval i (v: vect T i): Sym.type_of i→ X :=
match v with
| vnil ⇒(fun f ⇒ f)
| vcons u v ⇒(fun f ⇒ xeval v (f (eval u)))
end.
```

Fig. 3. Data-type for terms, and related evaluation function.

uninterpreted symbols (`e_bin`). This definition allows for a simple implementation of `eval`, given on the right-hand side. For uninterpreted symbols, the trick consists in using an accumulator to store the successive partial applications.

As expected, this syntax allows us to reify arbitrary user-level terms. For instance, take `(a∗S(b+b))−b`. We first construct the following environments where we store information about all atoms:

e_sym	e_bin
`1 ⇒ ⦇ ar := 1; value := S; compat := ... ⦈`	`1 ⇒ ⦇ value := plus; compat := ... ;`
`2 ⇒ ⦇ ar := 0; value := a; compat := ... ⦈`	`assoc := _ ; comm := Some ... ⦈`
`3 ⇒ ⦇ ar := 0; value := b; compat := ... ⦈`	`_ ⇒ ⦇ value := mult; compat := ... ;`
`_ ⇒ ⦇ ar := 2; value := minus; compat := ... ⦈`	`assoc := _ ; comm := None ⦈`

These environment functions are total: they associate a semantic value to indices that might be considered as "out-of-the-bounds". This requires environments to contain at least one value, but this makes the evaluation function total and easier to reason about: there is no need to return an option or a default value in case undefined symbols are encountered. We can then build a reified term whose evaluation in the above environments reduces to the starting user-level terms:

```
Let t := sym 4 ⟦bin_a 2 [(sym 2 ⟦⟧); (sym 1 ⟦bin_ac 1 [(sym 3 ⟦⟧,1);(sym 3 ⟦⟧,1)]⟧)]; sym 3 ⟦⟧⟧.
Goal eval e_sym e_bin t = (a∗S(b+b))−b. reflexivity. Qed.
```

Note that we cannot split the environment `e_bin` into two environments `e_bin_a` and `e_bin_ac`: since they would contain at least one binary operation with the proof that it is A or AC, it would not be possible to reify terms in a setting with only A or only AC operations. Moreover, having a single environment for all binary operations makes it easier to handle neutral elements (see §5.1).

Normalisation of reified terms in Coq. Normal forms are computed as follows: terms are recursively flattened under A/AC nodes and arguments of AC nodes are sorted. We give excerpts of this Coq function below, focusing on AC nodes: `bin_ac`' is a smart constructor that prevents building unary AC nodes, and `norm_msets norm i` normalises and sorts a multi-set, ensuring that none of its children starts with an AC node with index `i`.

```
Definition bin_ac' i (u: nemset T): T := match u with nil (u,1) ⇒ u | _ ⇒ bin_ac i u end.
Definition extract_ac i (s: T): nemset T :=
  match s with bin_ac j m when i = j ⇒ m | _ ⇒ [s,1] end.
Definition norm_msets norm i (u: nemset T): nemset T :=
  nefold_map merge_sort (fun (x,n) ⇒ copy_mset n (extract_ac i (norm x))) u
...
Fixpoint norm (u: T): T := match u with
| bin_ac i l ⇒ if is_commutative e_bin i then bin_ac' i (norm_msets norm i l) else u
| bin_a i l ⇒ bin_a' i (norm_lists norm i l)
| sym i l ⇒ sym i (vector_map norm l)
end.
```

Note that `norm` depends on the information contained in the environments: the look-up `is_commutative s_bin i` in the definition of `norm` is required to make sure that the binary operation `i` is actually commutative (remember that we need to store A and AC symbols in the same environment, so that we might have AC nodes whose corresponding operation is not commutative). Similarly, to handle neutral elements (§5.1), we will rely on the environment to detect whether some value is a unit for a given binary operation.

Correctness and completeness. We prove that the normalisation function is sound. This proof relies on the above defensive test against ill-formed terms: since invalid AC nodes are left intact, we do not need the missing commutativity hypothesis when proving the correctness of `norm`. We did not prove completeness. First, this is not needed to get a sound tactic. Second, this proof would be quite verbose (in particular, it requires a formal definition of equality modulo AC on reified terms). Third, we would not be able to completely prove the completeness of the resulting tactic anyway, since one cannot reason about the OCaml reification and normalisation functions in the proof-assistant [14,7].

3.2 Reification

Following the reflexive approach to solve an equality modulo AC, it suffices to apply the above theorem `decide` (§3.1) and to let Coq compute. To do so, we still need to provide two environments `e_bin` and `e_sym` and two terms `u` and `v`, whose evaluation is convertible to the starting user-level terms.

Type-class based reification. We do not want to rely on annotations (like projections of type-classes fields or canonical structures) to guess how to reify the terms: this would force the users to use our definitions and notations from the beginning. Instead, we let the users work with their own definitions, and we exploit type-classes to perform reification. The idea is to query the type-class resolution mechanism to decide whether a given subterm should be reified as an AC operation, an A operation, or an uninterpreted function symbol.

The inference of binary A or AC operations takes place first, by querying for instances of the classes `Associative` and `Commutative` on all binary applications. The remaining difficulty is to discriminate whether other applications should be considered as a function symbol applied to several arguments, or as a constant. For instance, considering the application `f (a+b) (b+c) c`, it suffices to query for `Proper` instances in the following order:

1.	`Proper` (R $\Rightarrow$ R $\Rightarrow$ R $\Rightarrow$ R)	(f)	?
2.	`Proper` (R $\Rightarrow$ R $\Rightarrow$ R)	(f (a+b))	?
3.	`Proper` (R $\Rightarrow$ R)	(f (a+b)(b+c))	?
4.	`Proper` (R)	(f (a+b)(b+c) c)	?

The first query that succeeds tells which partial application is a proper morphism, and with which arity. Since the relation `R` is reflexive, and any element is proper for a reflexive relation, the inference of constants—symbols of arity 0—is the catch-all case of reification. We then proceed recursively on the remaining arguments; in the example, if the second call is the first to succeed, we do not try to reify the first argument (`a+b`): the partial application `f(a+b)` is considered as an atom.

Reification language. We use OCaml to perform this reification step. Using the meta-language OCaml rather than the meta-language of tactics Ltac is a matter of convenience: it allows us to use more efficient data-structures. For instance, we use hash-tables to memoise queries to type-class resolution, which would have been difficult to mimic in Ltac or using canonical structures. The resulting code is non-trivial, but too technical to be presented here. Most of the difficulties come from the fact that we reify uninterpreted functions symbols using a dependently typed syntax, and that our reification environments contain dependent records: producing such Coq values from OCaml can be tricky. Finally, using Coq's plug-in mechanism, we wrap up the previous ideas in a tactic, `aac_reflexivity`, which automates this process, and solves equations modulo AC.

Efficiency. The dependently typed representation of terms we chose in order to simplify proofs does not preclude efficient computations. The complexity of the procedure is dominated by the merging of sorted multi-sets, which relies on a linear comparison function. We did not put this decision procedure through an extensive testing; however, we claim that it returns instantaneously in practice. Moreover, the size of the generated proof is linear with respect to the size of the starting terms. By contrast, using the tactic language to build a proof out of associativity and commutativity lemmas would usually yield a quadratic proof.

4 Matching modulo AC

Solving a matching problem modulo AC consists in, given a pattern p and a term t, finding a substitution σ such that $p\sigma \equiv_{AC} t$. There are many known algorithms [11,12,15,18]; we present here a simple one.

Naive algorithm. Matching modulo AC can easily be implemented non-deterministically. For example, to match a sum $p_1 + p_2$ against a term t, it suffices to consider all possible decompositions of t into a sum $t_1 + t_2$. If matching p_1 against t_1 yields a solution (a substitution), it can be used as an initial state to match p_2 against t_2, yielding a more precise solution, if any. To match a variable x against a term t, there are two cases depending on whether or not the

```
val (≫=): α m → (α → β m) → β m
val (≫|): α m → α m → α m
val return: α → α m
val fail: unit → α m
```

Fig. 4. Search monad primitives.

```
val split_ac: idx → term → (term ∗ term) m
val split_a : idx → term → (term ∗ term) m
```

Fig. 5. Search monad derived functions.

```
mtch (p1 +i p2) t σ = split_ac i t ≫= (fun (t1,t2) → mtch p1 t1 σ ≫= mtch p2 t2)
mtch (p1 ∗i p2) t σ = split_a  i t ≫= (fun (t1,t2) → mtch p1 t1 σ ≫= mtch p2 t2)
mtch (f(p̄)) (f(ū)) σ = fold_2 (fun acc p t → acc ≫= mtch p t) (return σ) p̄ ū
mtch (var x) t σ when Subst.find σ x = None = return (Subst.add σ x t)
mtch (var x) t σ when Subst.find σ x = Some v = if v ≡AC t then return σ else fail()
```

Fig. 6. Backtracking pattern matching, using monads.

variable has already been assigned in the current substitution. If the variable has already been assigned to a value v, we check that $v \equiv_{AC} t$. If this is not the case, the substitution must be discarded since x must take incompatible values. Otherwise, i.e., if the variable is fresh, we add a mapping from x to v to the substitution. To match an uninterpreted node $f(\overline{q})$ against a term t, it must be the case that t is headed by the same symbol f, with arguments $\overline{u}$; we just match $\overline{q}$ and $\overline{u}$ pointwise.

Monadic implementation. We use a monad for non-deterministic and backtracking computations. Fig. 4 presents the primitive functions offered by this monad: $\gg=$ is a backtracking bind operation, while $\gg|$ is non-deterministic choice. We have an OCaml type for terms similar to the inductive type we defined for Coq reified terms: applications of A/AC symbols are represented using their flattened normal forms. From the primitives of the monad, we derive functions operating on terms (Fig. 5): the function `split_ac i` implements the non-deterministic split of a term t into pairs (t_1, t_2) such that $t \equiv_{AC} t_1 +_i t_2$. If the head-symbol of t is $+_i$, then it suffices to split syntactically the multi-set of arguments; otherwise, we return an empty collection. The function `split_a i` implements the corresponding operation on associative only symbols. The matching algorithm proceeds by structural recursion on the pattern, which yields the code presented in Fig. 6 (using an informal ML-like syntax). A nice property of this algorithm is that it does not produce redundant solutions, so that we do not need to reduce the set of solutions before proposing them to the user.

Correctness. Following [11], we could have attempted to prove the correctness of this matching algorithm. While this could be an interesting formalisation work *per se*, it is not necessary for our purpose, and could even be considered an impediment. Indeed, we implement the matching algorithm as an oracle, in an arbitrary language. Thus, we are given the choice to use a free range of optimisations, and the ability to exploit all features of the implementation language. In any case, the prophecies of this oracle, a set of solutions to the matching problem, are verified by the reflexive decision procedure we implemented in §3.

```
Variable e_bin: idx → Bin.pack X R

Record binary_for (u: X) := {
 bf_idx: idx;
 bf_desc: Unit R (Bin.value (e_bin bf_idx)) u }.
```

```
Record unit_pack := {
 u_value: X;
 u_desc: list (binary_for u_value) }.

Variable e_unit: idx → unit_pack.
```

Fig. 7. Additional environment for terms with units.

5 Bridging the gaps

Combining the decision procedure for equality modulo AC and the algorithm for matching modulo AC, we get the tactic for rewriting modulo AC. We now turn to lifting the simplifying assumptions we made in the previous sections.

5.1 Neutral elements

Adding support for neutral elements (or "units") is of practical importance:

– to let `aac_reflexivity` decide more equations (e.g., `max` 0 (`b`∗1)+`a` = `a`+`b`);
– to avoid requiring the user to normalise terms manually before performing rewriting steps (e.g., to rewrite using $\forall x, x \cup x = x$ in the term $a \cap b \cup \emptyset \cup b \cap a$);
– to propose more solutions to pattern matching problems (consider rewriting $\forall xy, x \cdot y \cdot x^{\perp} = y$ in $a \cdot (b \cdot (a \cdot b)^{\perp})$, where $\cdot$ is associative only with a neutral element: the variable y should be instantiated with the neutral element).

Extending the pattern matching algorithm. Matching modulo AC with units does not boil down to pattern matching modulo AC against a normalised term: $a \cdot b \cdot (a \cdot b)^{\perp}$ is a normal form and the algorithm of Fig. 6 would not give solutions with the pattern $x \cdot y \cdot x^{\perp}$. The patch is however straightforward: it suffices to let the non-deterministic splitting functions (Fig. 5) use the neutral element possibly associated with the given binary symbol. For instance, calling `split_a` on the previous term would return the four pairs $\langle 1, a \cdot b \cdot (a \cdot b)^{\perp} \rangle$, $\langle a, b \cdot (a \cdot b)^{\perp} \rangle$, $\langle a \cdot b, (a \cdot b)^{\perp} \rangle$, and $\langle a \cdot b \cdot (a \cdot b)^{\perp}, 1 \rangle$, where 1 is the neutral element.

Extending the syntax of reified terms. An obvious idea is to replace non-empty lists (resp. multi-sets) by lists (resp. multi-sets) in the definition of terms—Fig. 3. This has two drawbacks. First, unless the evaluation function (Fig. 3) becomes a partial function, every A/AC symbol must then be associated with a unit (which precludes, e.g., `min` and `max` to be defined as AC operations on relative numbers). Second, two symbols cannot share a common unit, like 0 being the unit of both `max` and `plus` on natural numbers: we would have to know at reification time how to reify 0, is it an empty AC node for `max` or for `plus`? Instead, we add an extra constructor for units to the data-type of terms, and a third environment to store all units together with their relationship to binary operations. The actual definition of this third environment requires a more clever crafting than the other ones. The starting point is that a unit is nothing by itself, it is a unit for some binary operations. Thus, the type of the environment for units has to depend on the `e_bin` environment. This type is given in Fig. 7. The record `binary_for` stores a binary operation (pointed to by its index `bf_idx`) and a proof that the parameter `u` is a neutral element for this operation. Then, each unit is bundled with a list of operations it is a unit for (`unit_pack`): like for the environment `e_sym` , these dependent records allow us to use plain, non-dependent maps. In the end, the syntax of reified terms depends only on the environment for uninterpreted symbols (`e_sym`), to ensure that arities are respected, while the environment for units (`e_unit`) depends on that for binary operations (`e_bin`).

Extending the decision tactic. Updating the Coq normalisation function to deal with units is fairly simple but slightly verbose. Like we used the `e_bin` environment to check that `bin_ac` nodes actually correspond to commutative operations, we exploit the information contained in `e_unit` to detect whether a unit is a neutral element for a given binary operation. On the contrary, the OCaml reification code, which is quite technical, becomes even more complicated. Calling type-class resolution on all constants of the goal to get the list of binary operations they are a unit for would be too costly. Instead, we perform a first pass on the goal, where we infer all A/AC operations and for each of these, whether it has a neutral element. We construct the reified terms in a second pass, using the previous information to distinguish units from regular constants.

5.2 Subterms

Another point of high practical importance is the ability to rewrite in subterms rather than at the root. Indeed, the algorithm of Fig. 6 does not allow to match the pattern $x+x$ against the terms $f(a+a)$ or $a+b+a$, where the occurrence appears under some context. Technically, it suffices to extend the (OCaml) pattern matching function and to write some boilerplate to accommodate contexts; the (Coq) decision procedure is not affected by this modification. Formally, subterm-matching a pattern p in a term t results in a set of solutions which are pairs $\langle C, \sigma \rangle$, where C is a context and σ is a substitution such that $C[p\sigma] \equiv_{AC} t$.

Variable extensions. It is not sufficient to call the (root) matching function on all syntactic subterms: the instance $a + a$ of the pattern $x + x$ is not a syntactic subterm of $a + b + a$. The standard trick consists in enriching the pattern using a *variable extension* [19,21], a variable used to collect the trailing terms. In the previous case, we can extend the pattern into $y + x + x$, where y will be instantiated with b. It then suffices to explore syntactic subterms: when we try to subterm-match $x + x$ against $(a + c) * (a + b + a)$, we extend the pattern into $y + x + x$ and we call the matching algorithm (Fig. 6) on the whole term and the subterms a, b, c, $a + c$ and $a + b + a$. In this example, only the last call succeeds.

The problem with subterms and units. However, this approach is not complete in the presence of units. Suppose for instance that we try to match the pattern $x+x$ against $a*b$, where $*$ is associative only. If the variable x can be instantiated with a neutral element 0 for $+$, then the variable extension trick gives four solutions:

$$a * b + [] \qquad (a + []) * b \qquad a * (b + [])$$

(These are the returned contexts, in which [] denotes the hole; the substitution is always $\{x \mapsto 0\}$.) Unfortunately, if $*$ also has a neutral element 1, there are infinitely many other solutions:

$$a * b * (1 + []) \qquad a * b + 0 * (1 + []) \qquad a * b + 0 * (1 + 0 * (1 + [])) \qquad \ldots$$

(Note that these solutions are distinct modulo AC, they collapse to the same term only when we replace the hole with 0.) The latter solutions only appear when the pattern can be instantiated to be equal to a neutral element (modulo A/AC). We opted for a pragmatic solution in this case: we reject these peculiar solutions, displaying a warning message. The user can still instantiate the rewriting lemma explicitly, or make the appropriate transitivity step using `aac_reflexivity`.

6 Conclusions

The Coq library corresponding to the tools we presented is available from [9]. We do not use any axiom; the code consists of about 1400 lines of Coq and 3600 lines of OCaml. We conclude with related works and directions for future work.

6.1 Related Works

Boyer and Moore [8] are precursors to our work in two ways. First, their paper is the earliest reference to reflection we are aware of, under the name "Metafunctions". Second, they use this methodology to prove correct a simplification function for cancellation modulo A. By contrast, we proved correct a decision procedure for equality modulo A/AC with units which can deal with arbitrary function symbols, and we used it to devise a tactic for rewriting modulo A/AC.

Ring. While there is some similarity in their goals, our decision procedure is incomparable with the Coq `ring` tactic [14]. On the one hand, `ring` can make use of distributivity and opposite laws to prove goals like $\mathtt{x}^2 - \mathtt{y}^2 = (\mathtt{x}-\mathtt{y}) * (\mathtt{x}+\mathtt{y})$, holding in any ring structure. On the other hand, `aac_reflexivity` can deal with an arbitrary number of AC or A operations with their units, and more importantly, with uninterpreted function symbols. For instance, it proves equations like $\mathtt{f}(\mathtt{x} \cap \mathtt{y}) \cup \mathtt{g}(\emptyset \cup \mathtt{z}) = \mathtt{g}\ \mathtt{z} \cup \mathtt{f}(\mathtt{y} \cap \mathtt{x})$, where `f`, `g` are arbitrary functions on sets. Like with `ring`, we also have a tactic to normalise terms modulo AC.

Rewriting modulo AC in HOL and Isabelle. Nipkow [17] used the Isabelle system to implement matching, unification and rewriting for various theories including AC. He presents algorithms as proof rules, relying on the Isabelle machinery and tactic language to build actual tools for equational reasoning. While this approach leads to elegant and short implementations, what is gained in conciseness and genericity is lost in efficiency, and the algorithms need not terminate. The rewriting modulo AC tools he defines are geared toward automatic term normalisation; by contrast, our approach focuses on providing the user with tools to select and make one rewriting step efficiently.

Slind [21] implemented an AC-unification algorithm and incorporated it in the `hol90` system, as an external and efficient oracle. It is then used to build tactics for AC rewriting, cancellation, and modus-ponens. While these tools exploit pattern matching only, an application of unification is in solving existential goals. Apart from some refinements like dealing with neutral elements and A symbols, the most salient differences with our work are that we use a reflexive decision procedure to check equality modulo A/AC rather than a tactic implemented in the meta-language, and that we use type-classes to infer and reify automatically the A/AC symbols and their units.

Support for the former tool [17] has been discontinued, and it seems to be also the case for the latter [21]. To our knowledge, even though HOL-light and HOL provide some tactics to prove that two terms are equal using associativity and commutativity of a single given operation, tactics comparable to the ones we describe here no longer exist in the Isabelle/HOL family of proof assistants.

Rewriting modulo AC in Coq. Contejean [11] implemented in Coq an algorithm for matching modulo AC, which she proved sound and complete. The emphasis is put on the proof of the matching algorithm, which corresponds to a concrete implementation in the CiME system. Although decidability of equality modulo AC is also derived, this development was not designed to obtain the kind of tactics we propose here (in particular, we could not reuse it to this end). Finally, symbols can be uninterpreted, commutative, or associative and commutative, but neither associative only symbols nor units are handled.

Gonthier et al. [13] have recently shown how to exploit a feature of Coq's unification algorithm to provide "less ad hoc automation". In particular, they automate reasoning modulo AC in a particular scenario, by diverting the unification algorithm in a complex but really neat way. Using their trick to provide the generic tactics we discuss here might be possible, but it would be difficult. Our reification process is much more complex: we have uninterpreted function symbols, we do not know in advance which operations are AC, and the handling of units requires a dependent environment. Moreover, we would have to implement matching modulo AC (which is not required in their example) using the same methodology; doing it in a sufficiently efficient way seems really challenging.

Nguyen et al. [16] used the external rewriting tool ELAN to add support for rewriting modulo AC in Coq. They perform term rewriting in the efficient ELAN environment, and check the resulting traces in Coq. This allows one to obtain a powerful normalisation tactic out of any set of rewriting rules which is confluent and terminating modulo AC. Our objectives are slightly different: we want to easily perform small rewriting steps in an arbitrarily complex proof, rather than to decide a proposition by computing and comparing normal forms.

The ELAN trace is replayed using elementary Coq tactics, and equalities modulo AC are proved by applying the associativity and commutativity lemmas in a clever way. On the contrary, we use the high-level (but slightly inefficient) `rewrite` tactic to perform the rewriting step, and we rely on an efficient reflexive decision procedure for proving equalities modulo AC. (Alvarado and Nguyen first proposed a version where the rewriting trace was replayed using reflection, but without support for modulo AC [2].)

From the user interface point of view, leaving out the fact that the support for this tool has been discontinued, our work improves on several points: thanks to the recent plug-in and type-class mechanisms of Coq, it suffices for a user to declare instances of the appropriate classes to get the ability to rewrite modulo AC. Even more importantly, there is no need to declare explicitly all uninterpreted function symbols, and we transparently support polymorphic operations (like `List.app`) and arbitrary equivalence relations (like `Qeq` on rational numbers, or `iff` on propositions). It would therefore be interesting to revive this tool using the new mechanisms available in Coq, to get a nicer and more powerful interface.

Although this is not a general purpose interactive proof assistant, the Maude system [10], which is based on equational and rewriting logic, also provides an efficient algorithm for rewriting modulo AC [12]. Like ELAN, Maude could be used as an oracle to replace our OCaml matching algorithm. This would require some non-trivial interfacing work, however. Moreover, it is unclear to us how to use these tools to get all matching occurrences of a pattern in a given term.

6.2 Directions for Future works.

Heterogeneous terms and operations. Our decision procedure cannot deal with functions whose range and domain are distinct sets. We could extend the tactic to deal with such symbols, to make it possible to rewrite using equations like $\forall uv, \|u+v\| \leq \|u\| + \|v\|$, where $\|\cdot\|$ is a norm in a vector space. This requires a more involved definition of reified terms and environments to keep track of type information; the corresponding reification process seems quite challenging.

We could also handle heterogeneous associative operations, like multiplication of non-square matrices, or composition of morphisms in a category. For example, matrix multiplication has type $\forall$ `n m p`, `X n m` $\to$ `X m p` $\to$ `X n p` (`X n m` being the type of matrices with size n, m). This would be helpful for proofs in category theory. Again, the first difficulty is to adapt the definition of reified terms, which would certainly require dependently typed non-empty lists.

Other decidable theories. While we focused on rewriting modulo AC, we could consider other theories whose matching problem is decidable. Such theories include, for example, the Abelian groups and the Boolean rings [6] (the latter naturally appears in proofs of hardware circuits).

Integration with other tools. Recently, tactics have been designed to exploit external SAT/SMT solvers inside Coq [3]. These tactics rely on a reflexive proof checker, used to certify the traces generated by the external solver. However, in the SMT case, these traces do not contain proofs for the steps related to the considered theories, so that one needs dedicated Coq decision procedures to validate these steps. Currently, mostly linear integer arithmetic is supported [3], using the `lia` tactic [5]; our tactic `aac_reflexivity` could be plugged into this framework to add support for theories including arbitrary A or AC symbols.

Acknowledgements

We would like to thank Matthieu Sozeau for his precious help in understanding Coq's internal API.

References

1. S. F. Allen, R. L. Constable, D. J. Howe, and W. E. Aitken. The Semantics of Reflected Proof. In *Proc. LICS*, pages 95–105. IEEE Computer Society, 1990.
2. C. Alvarado and Q.-H. Nguyen. ELAN for Equational Reasoning in Coq. In *Proc. LFM'00*. INRIA, 2000. ISBN 2-7261-1166-1.
3. M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In *Proc. CPP*, LNCS. Springer, 2011. To appear in this volume.
4. G. Barthe, M. Ruys, and H. Barendregt. A Two-Level Approach Towards Lean Proof-Checking. In *Proc. TYPES*, LNCS, pages 16–35. Springer, 1995.
5. F. Besson. Fast reflexive arithmetic tactics the linear case and beyond. In *Selected papers of TYPES'06*, volume 4502 of *LNCS*, pages 48–62. Springer, 2007.
6. A. Boudet, J.-P. Jouannaud, and M. Schmidt-Schauß. Unification in Boolean Rings and Abelian groups. *J. Symb. Comput.*, 8(5):449–477, 1989.
7. S. Boutin. Using Reflection to Build Efficient and Certified Decision Procedures. In *Proc. TACS*, volume 1281 of *LNCS*, pages 515–529. Springer, 1997.
8. R. S. Boyer and J. S. Moore, editors. *The Correctness Problem in Computer Science*. Academic Press, 1981.
9. T. Braibant and D. Pous. Tactics for working modulo AC in Coq. Coq library available at `http://sardes.inrialpes.fr/~braibant/aac_tactics/`, June 2010.
10. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 system. In *Proc RTA*, volume 2706 of *LNCS*. Springer, 2003.
11. E. Contejean. A Certified AC Matching Algorithm. In *Proc. RTA*, volume 3091 of *LNCS*, pages 70–84. Springer, 2004.
12. S. Eker. Single Elementary Associative-Commutative Matching. *J. Autom. Reasoning*, 28(1):35–51, 2002.
13. G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. How to make ad hoc proof automation less ad hoc. In *Proc. ICFP*. ACM, 2011. To appear.
14. B. Grégoire and A. Mahboubi. Proving equalities in a commutative ring done right in Coq. In *Proc. TPHOLs*, volume 3603 of *LNCS*, pages 98–113. Springer, 2005.
15. J. M. Hullot. Associative Commutative pattern matching. In *Proc. IJCAI*, pages 406–412. Morgan Kaufmann Publishers Inc., 1979.
16. Q. H. Nguyen, C. Kirchner, and H. Kirchner. External Rewriting for Skeptical Proof Assistants. *J. Autom. Reasoning*, 29(3-4):309–336, 2002.
17. T. Nipkow. Equational reasoning in Isabelle. *Sci. Comp. Prg.*, 12(2):123–149, 1989.
18. T. Nipkow. Proof transformations for equational theories. In *Proc. LICS*, pages 278–288. IEEE Computer Society, 1990.
19. G. Peterson and M. Stickel. Complete sets of reductions for some equational theories. *J. ACM*, 28(2):233–264, 1981.
20. G. Plotkin. Building in equational theories. *Machine Intelligence 7*, 1972.
21. K. Slind. AC Unification in HOL90. In *Proc. HUG*, volume 780 of *LNCS*, pages 436–449. Springer, 1993.
22. M. Sozeau and N. Oury. First-class type classes. In *Proc. TPHOL*, volume 4732 of *LNCS*, pages 278–293. Springer, 2008.