A Tactic for Deciding Kleene Algebras

Thomas Braibant, Damien Pous

To cite this version:

Thomas Braibant, Damien Pous. A Tactic for Deciding Kleene Algebras. 2009. hal-00383070v1

HAL Id: hal-00383070
https://hal.science/hal-00383070v1

Preprint submitted on 11 May 2009 (v1), last revised 20 May 2011 (v5)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Tactic for Deciding Kleene Algebras

Thomas Braibant
ENS Lyon – INRIA

Damien Pous
CNRS

Abstract

We present a Coq reflexive tactic for deciding equalities or inequalities in Kleene algebras. This tactic is part of a larger project, whose aim is to provide tools for reasoning about binary relations in Coq: binary relations form a Kleene algebra, where the *star* operation is the reflexive transitive closure. Our tactic relies on an initiality theorem by Kozen, whose proof goes by replaying finite automata algorithms in an algebraic way, using matrices.

Motivations

Proof *assistants* like Coq make it possible to leave technical or administrative details to the computer, by defining high-level tactics. For example, one can define tactics in order 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 inequalities in Kleene algebras. This corresponds to a broader goal: providing tools (tactics) for working with binary relations. Indeed, Kleene algebras correspond to a non-trivial decidable fragment of binary relations. In the long term, we plan to use these tools for formalising 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 thrice. Informal diagrams are drawn on the left. The goal listed in the middle corresponds to a naive formalisation where the points related by 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), then the diagrams corresponding to a given state in the proof can easily be read from Coq's output.

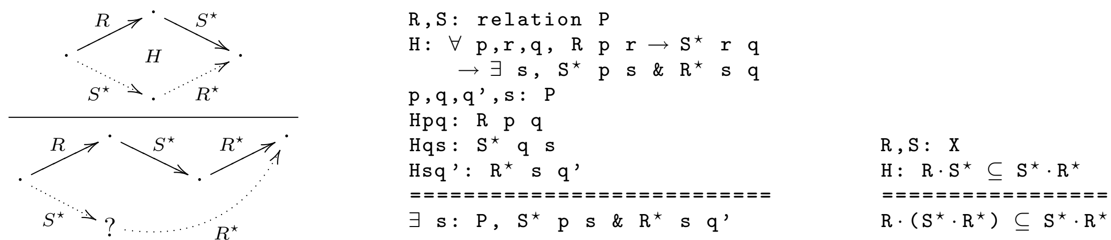

Figure 1: Diagrammatic, concrete, and abstract presentations of the same state in a proof.

Some technology is then required to avoid handling some administrative steps explicitly, which were somehow hidden in concrete proofs contexts. For example, in the right-hand side proof state of Fig. 1, we first need to re-arrange parentheses in order to be able to rewrite the goal using hypothesis H. This drawback is eliminated by defining adequate tactics to work modulo associativity and commutativity.

More importantly, moving to the abstract setting allows us to implement several decision procedures that could hardly be stated with the concrete presentation. For example, once we rewrite H in the right-hand side goal of Fig. 1, we obtain the inclusion $\mathrm{S}^\star{\cdot}\mathrm{R}^\star{\cdot}\mathrm{R}^\star{\subseteq}\mathrm{S}^\star{\cdot}\mathrm{R}^\star$ which is a (straightforward) theorem of Kleene algebras: it can be proved automatically thanks to the tactic we describe in this paper.

Outline. We recall the required mathematical background and we sketch the structure of the tactic in Sect. 1. We give some details about the underlying design choices in Sect. 2. Sect. 3 focuses on the algebraic part of the correctness proof, and the implemented algorithms are described in Sect 4. We conclude with directions for future work in Sect. 5.

1 Deciding equalities in Kleene algebras

Theoretical background. A Kleene algebra [18] 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 operation on X, satisfying the following axiom and inference rules (where $\leq$ is the preorder defined by $x \leq y \triangleq x + y = y$):

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

Models of Kleene algebras include *regular languages*, where the star operation is language iteration; and *binary relations*, where the product $(\cdot)$ is relational composition, and star is reflexive and transitive closure (in this model, the above rules basically state that $a^\star$ is the least reflexive element which is stable under composition with a).

Thanks to finite automata theory (among others, Kleene [17], Rabin & Scott [23], Nerode [22]), equality of regular languages is decidable:

> *"two regular expressions denote the same regular language if and only if the corresponding minimal automata are isomorphic",*

and minimal automata can be computed as follows: 1) construct a non-deterministic finite automaton with epsilon-transitions (ϵ-NFA), by structural induction on the regular expression; 2) remove epsilon-transitions to obtain a non-deterministic finite automaton (NFA), by computing the closure of epsilon-transitions; 3) determinise the automaton using the accessible subsets construction, to obtain a deterministic finite automaton (DFA); 4) minimise the DFA by merging all states that are equivalent according to Myhill-Nerode's relation.

However, the above theorem is not sufficient to decide equality in all Kleene algebras: it only applies to the regular languages model. We actually need a more recent theorem, by Kozen [18]:

> *"if two regular expressions α and β denote the same regular language, then $\alpha = \beta$ can be proved in any Kleene algebra".*

In other words, the algebra of regular languages is initial among Kleene algebras: we can use the above decision procedure to solve equations in an arbitrary Kleene algebra $\mathcal{A}$. The main idea of Kozen's proof is to encode automata using matrices over $\mathcal{A}$, and to replay automaton

algorithms at this algebraic level. Indeed, a finite automaton with transitions labelled by the elements of $\mathcal{A}$ can be represented with three matrices $(u, M, v) \in \mathcal{M}_{1,n} \times \mathcal{M}_{n,n} \times \mathcal{M}_{n,1}$: 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: $M_{i,j}$ is non-empty if there is a transition from state i to state j. This corresponds to the definition of an ϵ-NFA; definitions of NFAs and DFAs can easily be recovered by adding conditions on matrices u and M.

We remark that the product $u \cdot M \cdot v$ is a scalar, which can be thought of as the set of one-letter words accepted by the automaton. Therefore, in order to mimic the actual behaviour of a finite automaton, we just need to iterate over the matrix M. This is possible thanks to another theorem by Kozen, which actually is the crux of its initiality theorem: *"square 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 $u \cdot M^\star \cdot v$. In the regular languages model, this expression actually corresponds to the language recognised by the automaton. We give more details about this proof in Sect. 3.

Overview of our strategy. We define a *reflexive* tactic. This methodology is quite standard: it is described in [1] and it was used by Grégoire and Mahboubi to obtain the current `ring` tactic [12]. Concretely, this means that we implement the decision as a Coq program (Sect. 4), so as to be able to prove its correctness within the proof assistant (Sect. 3):

```
Definition decide_Kleene: regexp → regexp → bool := ...
Theorem Kozen: ∀ a,b: regexp, decide_Kleene a b = true → a ≐ b.
```

The above statement corresponds to Kozen's theorem in the special case of the "free Kleene algebra": `regexp` is the obvious inductive type for regular expressions over a given set of variables, and $\doteq$ is the inductive equality generated by the axioms of Kleene algebras. Using Coq's reification mechanism, this is sufficient for our needs: the result can be lifted to other models using simple tactics (we return to this point in Sect. 2.3).

The equational theory of Kleene algebras is PSPACE-complete [20, 21]. Indeed, the determinisation phase of the algorithm we sketched above can produce automata of exponential size. Although this is not the case on the typical examples we tried, where our tactic runs almost instantaneously, this means that the `decide_Kleene` function must be written with some care, using efficient out-of-the-shelf automaton algorithms. Notably, the matricial representation of automata is not efficient for all stages of the decision procedure. Therefore, we need to work with other data-structures for automata, and to write the corresponding translation functions in order to reason about the algorithms in the uniform setting of matricial automata. We detail and justify our choices about these algorithms and data structures in Sect. 4.

2 Underlying design choices

Before going through Kozen's proof (Sect. 3) and giving details about our implementation of the decision procedure (Sect. 4), we explain the main choices we made about the structure of our development: how to represent the algebraic hierarchy, how to represent matrices, how to manage matrix dimensions, and how to resort to syntactical objects using reification.

2.1 Algebraic hierarchy

The mathematical definition of a Kleene algebra is incremental: it is a non-commutative semiring, which is itself composed of a monoid and a semi-lattice. Moreover, proofs naturally follow this hierarchy: when proving results about semirings, one usually rely on results about both

monoids and semi-lattices. In order to structure our development in a similar way, we defined the algebraic hierarchy using Coq's recent *typeclasses* mechanism [24]: we defined several classes, corresponding to the different algebraic structures, so as to obtain the following "sub-typing" relations (these relations are projections, declared as morphisms to the typeclass system):

```
SemiLattice <:
  Monoid    <:  SemiRing <: KleeneAlgebra <: ...
```

The other possibilities were to use *canonical structures* or *modules*. We tried the latter one; it was however quite difficult to organise modules, signatures and functors so as to obtain the desired level of sharing between the various proofs. In particular, when we consider more complex algebraic structures, we can no longer work with syntactical sub-typing between structures (we only have functors from one structure to another) and we lose the ability to directly use theorems, definitions, and tactics from lower structures in higher structures.

Except for some limitations due to the novelty of this feature, typeclasses happen to be much easier to use for our purposes: sharing is obtained in a straightforward way, the code does not need to be written in a monolithic way (as opposed to using functors), and it brings nice solutions for overloading notations (e.g., we can use the same infix symbol for multiplication in a monoid, a semiring, or a matrix semiring). We currently try to compare our strategy with that from [11], which is based on canonical structures. Although the aims of canonical structures and typeclasses are quite close, the underlying mechanisms lead to different constraints.

2.2 Matrices

Coq definition. A matrix can be seen as a partial map from pairs of integers to a given type X, so that a Coq definition of matrices and the sum operation could be the following:

```
Definition MX (n m: nat) := ∀ i j, i<m → j<n → X.
Definition plus n m (M N: MX n m) i j (Hi: i<n) (Hj: j<n) := 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 that they use vectors of vectors, this is the approach followed by [11] and [4]. With such a representation, every access to a matrix elements must be made by exhibiting two proofs, ensuring that the indices lie within the bounds. For simple operations like the above `sum` function this is not so problematic, this however becomes quite a burden when writing more complex operations like matrix multiplication. We actually chose to move these bounds checks to equality proofs only, by working with the following definitions:

```
Definition MX n m := nat → nat → X.
Definition equal n m (M N: MX n m) := ∀ i j, i<n → j<m → M i j = N i j.
Definition dot n m p (A: MX n m) (B: MX m p) := fun i j ⇒ sum 0 m (fun k ⇒ M i k · N k j).
```

Here, a matrix is an infinite function from pairs of integers to X, and equality is restricted to the domain of the matrix. With these definitions, we do not need to manipulate proofs when defining matrix operations (like the above `dot` function), so that these definitions are both easier to write and more efficient to compute with. Bounds checks are required a posteriori only, when proving properties about these matrices operations, e.g., associativity of the product. This is easy in most cases: these proofs are done within the interactive proof mode, and can often be solved with high level tactics like `omega`. We have not yet found drawbacks to this approach, we do not know whether it scales to more intensive usages like linear algebra [11].

Phantom types. Unfortunately, these definitions allow one to type the following code:

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

```
X: Type.                              T: Type.
                                      X: T → T → Type.

dot:  X → X → X.                      dot:  ∀ A B C, X A B → X B C → X A C.
one:  X.                              one:  ∀ A,     X A A.
plus: X → X → X.                      plus: ∀ A B,   X A B → X A B → X A B.
zero: X.                              zero: ∀ A B,   X A B.
star: X → X.                          star: ∀ A,     X A A → X A A.

dot_neutral_left:                     dot_neutral_left:
 ∀ x, dot one x = x.                   ∀ A B (x: X A B), dot one x = x.
...                                   ...
```

Figure 2: From Kleene algebras to typed Kleene algebras.

This definition is accepted by Coq because of the conversion rule: since `MX n m` is a dependent type that does not mention `n` nor `m` in its body, these type informations can be discarded by the Coq type system, using the conversion rule (`MX n 16 = MX 64 p`). This is not so terrible: such an ill-formed definition will be detected at proof-time. It is however a bit sad not to benefit from the advantages of a strongly typed programming language here. We partially solved this problem 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: nat) (M: MX n m) := match f with box f ⇒ f end.
Definition plus (n m: nat) (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 64 p` 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`). However, we need to artificially introduce the `box` and `get` functions at each matrix construction or access, which is slightly inefficient. We still look for a better solution to this problem.

Computation. From a computational point of view, using lazy functions as a representation for matrices is two-edged : on the one hand, if the resulting matrix of a computation is seldom used, then computing the result point-wise, by need, is efficient; on the other hand, making numerous accesses to the same expensive computation may be a burden. Therefore, we have defined a *memoisation* operator that computes every point of a matrix, store the result in an associative map, and returns a function (of the same type) that accesses the associative map instead of recomputing the result. Since this memoisation operator can be proved to be an identity, it can be inserted in our code in a transparent way, at judicious places.

```
Lemma mx_force_id : ∀ n m (M : MX n m), mx_force M = M.
```

2.3 Typed algebras, typed reification

Adding types. *Square* matrices over a semiring form a semiring, and Kozen needed to extend this folklore result to Kleene algebras [18]. For *rectangular* matrices, the various operations are only partial: dimensions have to agree. Therefore, with naive definitions of the algebraic structures, we are unable to use theorems and tools developed for monoids, semi-lattices, and semirings to reason about rectangular matrices. To remedy this problem, we generalised algebraic structures from the beginning, using *types*. An example is given in Fig. 2: a typical signature for semirings is presented on the left-hand side; we moved to the signature on the right-hand side, where a set `T` of types is used to constrain the various operations. These types can be thought of as matrix dimensions; we can also remark that we actually moved to a categorical setting: `T` is a set of objects, `X A B` is the set of morphisms from `A` to `B`, `one` is the set of identities, and `dot` is composition. As expected, with such definitions, one can form arbitrary matrices over a typed structure, and obtain another instance of this typed structure:

```
Instance mx_SemiRing: SemiRing → SemiRing := ...
Instance mx_KleeneAlgebra: KleeneAlgebra → KleeneAlgebra := ...
```

Then, thanks to typeclasses, we inherit all theorems, tactics, and notations we defined on generic structures, at the matricial level. Notably, when defining the star operation on matrices over a Kleene algebra, we can benefit from all tools for semirings, monoids, and semi-lattices, at the matricial level. This is quite important since this construction is rather complicated.

Removing types. Typed structures not only make it easier to work with matrices, they also give rise to a wider range of models. In particular, we can consider heterogeneous binary relations (between two distinct sets), rather than binary relations on a fixed set. This leads to the following question: can the usual decision procedures (for semi-lattices, semirings, and the one presented here for Kleene algebras) be extended to this more general setting?

Consider for example the equation $a \cdot (b \cdot a)^\star = (a \cdot b)^\star \cdot a$, which is a theorem of typed Kleene algebras as soon as a and b are respectively given types $A \to B$ and $B \to A$, for some A, B; how to make sure that the proof obtained by computing minimal (untyped) automata and concluding using Kozen initiality theorem is actually a valid, well-typed, proof?

For efficiency and practicability reasons, re-defining our decision procedures to work with typed objects is not an option (they are written as reflexive tactics). Instead, we managed to prove the following theorem, which allows one to erase types, i.e., to transform a typed equality goal into an untyped one:

$$\frac{T\Sigma \vdash u = v \qquad \Gamma \vdash u \rhd \alpha : A \to B \qquad \Gamma \vdash v \rhd \beta : A \to B}{\mathcal{A} \vdash \alpha = \beta : A \to B} \tag{$*$}$$

Here, $\Gamma \vdash u \rhd \alpha : A \to B$ reads "under the evaluation and typing context Γ, the untyped term u can be evaluated to α, of type $A \to B$"; this predicate can be defined inductively in a straightforward way, for various algebraic structures. The theorem can then be rephrased as follows: "given an untyped equality proof of u and v, and typed interpretations α and β for u and v, we can construct a typed proof of $\alpha = \beta$". We proved it for semi-lattices, monoids, semirings, and Kleene algebras, so that all of our decision tactics apply to the typed setting – and in particular, to matrices. While this theorem is trivial for semi-lattices, and rather simple for monoids, difficulties arise with semirings and Kleene algebras, due to the presence of annihilator elements. Also note that Kozen investigated a similar question [19] and came up with a slightly different solution: he solves the case of the Horn theory rather than 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.

Typed reification. The above discussion about types raises another issue: reflexive tactics need to work with syntactical objects. For example, in order to construct an automaton, we need to proceed by structural induction on the given expression. This step is commonly achieved by moving to the free algebra of terms, and resorting to Coq's reification mechanism (`quote`). However, this mechanism does not handle typed structures, so that we needed to re-implement it. Since we do not have binders, we were able do this within `Ltac`: it suffices to `eapply` theorem ($*$) to the current goal, so that we are left with three goals, with holes for u, v and Γ; then by using an adequate representation for Γ, and by exploiting the very simple form of the typing and evaluation predicate, we are able to progressively fill these holes and to close the two goals about evaluation by repeatedly applying constructors and ad-hoc lemmas about environments. Unlike Coq's standard `quote`, which works by conversion and has no impact on the size of the current proof, this "lightweight"-`quote` generates rather large proof-terms. We would like to understand whether this situation can be improved, still remaining within `Ltac`.

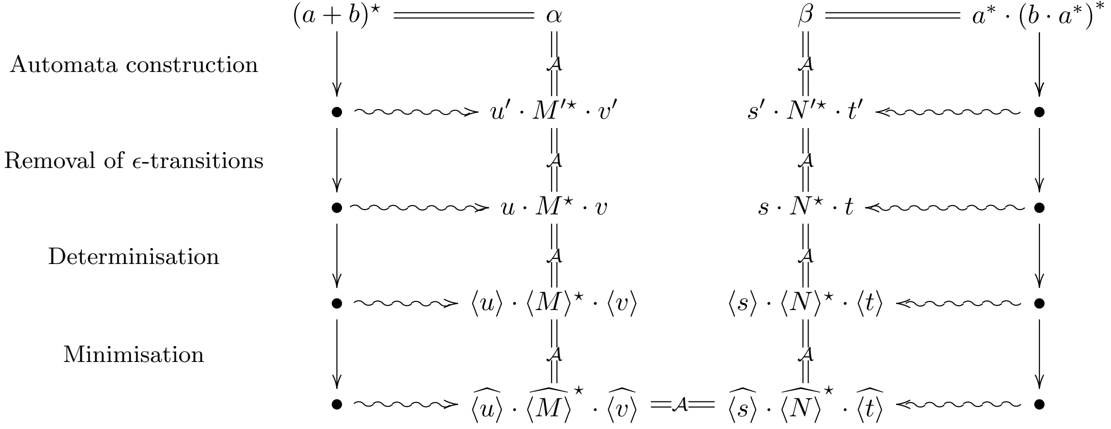

Figure 3: Soundness of `decide_Kleene`.

3 Kozen's proof

The tactic we describe here relies on Kozen's initiality theorem: to prove that an equality $\alpha = \beta$ holds in any Kleene algebra, it suffices to check that the underlying minimal automata are isomorphic. The overall structure of Kozen's proof is depicted on Fig. 3: bullets represent idealised standard automaton constructions; the proof consists in showing that each construction can be related to a matricial automaton, whose interpretation is provably equal to the initial expression; we finally conclude by transitivity, if the minimal automata coincide. We briefly sketch the inner steps of this proof, i.e., the algebraic part, letting the reader refer to [18] for more details. The algorithms corresponding to the outer arrows are described in Sect. 4.

Building automata. There are several ways of constructing an ϵ-NFA from a regular expression [27]. We chose Thomson's construction [26] because of its simplicity: as described in [18], this is only a matter of block matrix constructions, and we easily show that the ϵ-NFA built from α evaluates to α, using algebraic laws. For example, the automaton for a sum is defined, and proved correct, as follows; the other constructions are obtained in a very similar way.

$$\left[\begin{array}{c|c} u & s \end{array}\right] \cdot \left[\begin{array}{c|c} M & 0 \\ \hline 0 & N \end{array}\right]^\star \cdot \left[\begin{array}{c} v \\ \hline t \end{array}\right] = \left[\begin{array}{c|c} u & s \end{array}\right] \cdot \left[\begin{array}{c|c} M^\star & 0 \\ \hline 0 & N^\star \end{array}\right] \cdot \left[\begin{array}{c} v \\ \hline t \end{array}\right] = \cdots = u \cdot M^\star \cdot v + s \cdot N^\star \cdot t$$

While these constructions are rather simple, they heavily rely on block matrix properties. The fact that we do not use dependent types to represent matrices greatly helps here.

Removing ϵ-transitions. The automata obtained with Thomson's construction may contain ϵ-transitions: their transitions matrices can be written as $M = J + \sum_{a\in\Sigma} a \cdot N_a$, where J and the N_a are 0-1 matrices, and J corresponds to the graph of ϵ-transitions. Removing these transitions to obtain an NFA usually means computing their reflexive and transitive closure, to update the other transitions. This can be done algebraically: thanks to the identity $(a + b)^\star = a^\star \cdot (b \cdot a^\star)^\star$ (a theorem of Kleene algebras), we have $u \cdot (J + N)^\star \cdot v = u \cdot J^\star \cdot (N \cdot J^\star)^\star \cdot v$, and the automaton on the right $(u \cdot J^\star, N \cdot J^\star, v)$ no longer contains ϵ-transitions. Indeed, $J^\star$ corresponds to the reflexive transitive closure of J.

Determinisation. The determinisation algorithm we implemented builds a DFA whose states are sets of states from the initial NFA; it consists in enumerating the set of subsets of states

that are accessible from the set of initial states. Starting from a NFA (u, M, v) with n states, this algorithm returns a DFA $(\langle u\rangle, \langle M\rangle, \langle v\rangle)$ with $\langle n\rangle$ states, together with a map ρ from $[1..\langle n\rangle]$ to the subsets of $[1..n]$. We sketch the algebraic part of the correctness proof. By letting X denote the $(\langle n\rangle, n)$ 0-1 matrix defined by $X_{sj} \triangleq j \in \rho(s)$, we prove that the returned automaton satisfies the following commutation properties:

$$\langle M\rangle \cdot X = X \cdot M \quad (1) \qquad\qquad \langle u\rangle \cdot X = u \quad (2) \qquad\qquad \langle v\rangle = v \cdot X \quad (3)$$

The intuition behind X is that this is a "decoding" matrix: it sends the characteristic vectors of states of the DFA to the characteristic vectors of the corresponding subset of states from the NFA. Therefore, (1) can be read as follows: executing a transition in the DFA and then decoding the result amounts to decoding the given 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. From (1), we can deduce $\langle M\rangle^\star \cdot X = X \cdot M^\star$ using a theorem of Kleene algebras, and we can conclude with $(2, 3)$: the two automata evaluate to the same value:

$$\langle u\rangle \cdot \langle M\rangle^\star \cdot \langle v\rangle = \langle u\rangle \cdot \langle M\rangle^\star \cdot X \cdot v = \langle u\rangle \cdot X \cdot M^\star \cdot v = u \cdot M^\star \cdot v \ .$$

Minimisation. The algebraic part of the correctness proof for minimisation is similar to that for determinisation. Starting from a DFA (u, M, v), the algorithm computes a partition of states, such that equivalence classes are stable under transitions and refine the partition of states between final and non-final. This partition is computed using Hopcroft's algorithm, which is described in Sect. 4; it is then converted into a map $[.]$ sending each state of the given DFA to the canonical representant of its equivalence class. This map allows us to define a decoding matrix Y by letting $Y_{ij} \triangleq [i] = j$, and the minimised automaton $(\widehat{u}, \widehat{M}, \widehat{v})$ is defined by:

$$\widehat{M} \triangleq Y^\top \cdot M \cdot Y \qquad\qquad \widehat{u} \triangleq u \cdot Y \qquad\qquad \widehat{v} \triangleq Y^\top \cdot v \ .$$

We finally prove that $Y \cdot \widehat{M} = M \cdot Y$ and $Y \cdot \widehat{v} = v$ (the first equality means that merging equivalence classes and then computing transitions in the minimised automaton amounts to computing transitions in the initial automaton and then merging the resulting states). As previously, this yields $\widehat{u} \cdot (\widehat{M})^\star \cdot \widehat{v} = u \cdot M^\star \cdot v$: the automata are equivalent.

4 Implementing the decision procedure

We now focus on the external part of Fig. 3, that is, the algorithmic details of our implementation. As explained in the introduction, the equational theory of Kleene algebras being PSPACE-complete, we have to care about efficiency. This drives our choices about both data-structures and algorithms: accessible subset construction for determinisation, and Hopcroft's minimisation algorithm [15] (which runs in $O(n \log n)$ where n is the size of the input DFA).

Boolean matrices. It is inefficient to work with matrices over the free Kleene algebra: there are many terms that will never appear in automata matrices (like $(a+1)^\star$), and we must reason up to the axioms of Kleene algebra, that equate, e.g., $1+a$ and $0^\star+(1+0\cdot b)\cdot a$. Since Thomson's construction yields automata whose transition matrix can be written as $M = J + \sum_{a\in\Sigma} a \cdot N_a$, where J and the N_a are 0-1 matrices, it actually suffices to work with matrices built upon the Kleene algebra of booleans: $\langle$bool,andb,orb,true, false ,fun _⇒true$\rangle$. Then, in proofs, we inject these matrices into those built upon the free Kleene algebra (e.g., for evaluating automata formally). This allows us to write optimised functions for boolean matrices: there are only two values to consider and we can exploit laziness. In particular, removing ϵ-transitions can be done

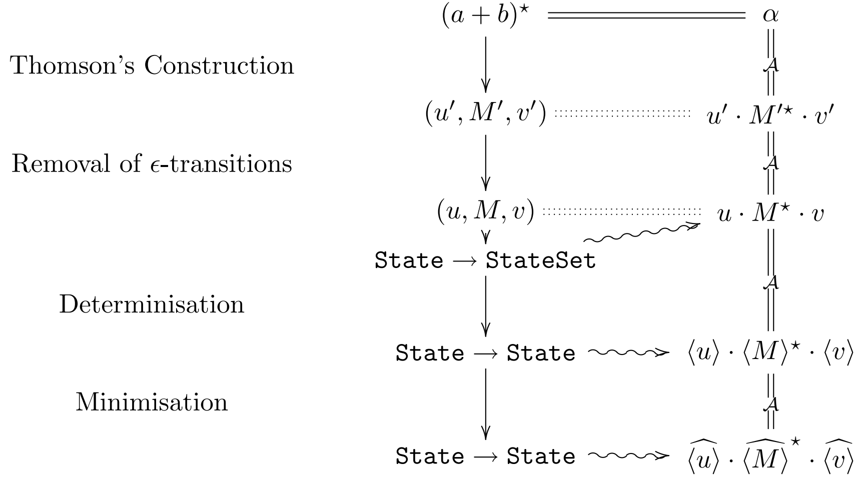

Figure 4: Proof and code relationship.

easily and efficiently with this representation of automata: as showed in Sect. 3, it suffices to compute the star of a boolean matrix ($J^\star$), and some multiplications ($u \cdot J^\star$ and the $N_a \cdot J^\star$).

FSets. The matricial representation of automata is no longer adequate when it comes to determinisation and minimisation. Therefore, starting from the ϵ-free NFA we built, we convert our matrices to more convenient representations, like transition functions. As can be seen on Fig. 4, we start with non-deterministic transition functions that map states to sets of states, and we later use deterministic transition functions that map states to states. To build efficiently these functions, we use the finite sets and finite maps libraries of Coq to represent state sets, partitions of states, and so on… These libraries being rather complete, this also gives us proper tools for proving the correction of our algorithms, and linking these structures to the matricial representation of automata (the horizontal arrows from Figs. 3 and 4).

Determinisation. Determinisation is exponential in worst case: this is a power-set construction. However, examples where this bound is reached are rather contrived, and the practical complexity is much better: most subsets of states cannot be reached from the subset of initial states. It is therefore crucial to implement the accessible subset construction, so as to avoid useless computations. We only give a very high level view of our implementation here: the standard algorithm is basically a while loop; that we translate into a tail-recursive fix-point; termination is not structural: it requires us to compute the exponential worst case bound, and we use a standard trick in order to avoid this useless and problematic computation. The proof of the algorithm requires us to find the adequate invariant for the loop; due to tail-recursion, this rather large invariant cannot be defined progressively with Coq's help: it has to be defined by hand, in a monolithic way.

Minimisation. We have to compute the Myhill-Nerode equivalence relation, which equates states sharing the same behaviour, i.e., accepting the same the language. Our implementation of Hopcroft's algorithm [14, 15] is given in Fig. 5: it consists in a 'while' loop containing two nested 'for' loops, translated using the `fold` operation of finite sets. Again, termination is not structural and imposes us to use well-founded recursion with a lexicographic order.

```
(* a           : label                      *)
(* i           : state                      *)
(* p,q,pt,pf   : fset state                 *)
(* P           : fset (fset state)          *)
(* L           : fset (label * fset state)  *)

Variables states, finaux: fset state.
Variable  labels: fset label.
Variable  delta: state → label → fset state.

Definition splittable p a q :=
  let (pt,pf) :=
    partition (fun i ⇒ (delta i a) ∈ q) p
  in if is_empty pt || is_empty pf
    then None
    else Some (pt,pf).

Definition update_splitters p pf pt L :=
  fold (fun a L ⇒ if (a,p) ∈ L
    then {(a,pf),(a,pt)} ∪ L\(a,p)
    else if cardinal pf < cardinal pt
      then {(a,pf)} ∪ L
      else {(a,pt)} ∪ L
  ) labels L.
```

```
Definition split P L (a,q) :=
  fold (fun p acc ⇒
    match splittable p a q with
      | None ⇒ acc
      | Some (pf,pt) ⇒
        let (P,L) := acc in
          ({pf, pt} ∪ P\p,
           update_splitters p pf pt L))
    end
  ) P (P,L).

Function loop P L {wf RPL (P,L)} :=
  match choose L with
    | None ⇒ P
    | Some x ⇒ loop (split P (L\x) x)
  end.

Definition partition :=
  loop
    {finals, states\finals}
    (labels × {finals}).
```

Figure 5: Coq code for minimisation.

The idea of the algorithm is to start from an initial partition of states (final and non final states), and to refine this partition whenever a state is `splittable`: i.e., when a move from a set of state can lead to two different sets by a transition with a given label `a`. Hopcroft uses a set `L` of *splitters*, i.e., pairs (label, state set) w.r.t. which one must attempt to split classes of the partition. The crux of the algorithm is to keep from adding too much redundancy in `L`: if a pair `(a,q)` is not in this set, then either every class of the partition is already split w.r.t. `(a,q)`, or `L` contains enough pairs to *subsume* `(a,q)`. Surprisingly, we did not find any precise account of this subsumption relation in the literature, although it seems to be at the heart of the correctness proof: it is part of the invariant for the 'while' loop.

Treading through `L` in the main `loop` function, we dismiss the pairs `(a,q)` that do not split equivalence classes, and we update our partition `P` and the set `L` when `(a,q)` splits an equivalence class `p` into `pf` and `pt`. The update of potential splitters in `L` is based on the following remark: when `p` is split into `pf, pt`, then, for any label `a`, it suffices to split every other class `q` w.r.t. any two of `(a,p)`, `(a,pf)`, and `(a,pt)`. If `(a,p)` $\in$ `L`, we must add both sub-splitters; if `(p,a)` $\notin$ `L`, then `L` subsumes `(p,a)` and it suffices to add the smallest of `(a,pf)` and `(a,pt)` to `L`. At the end of the algorithm, since `L` is empty, we know that the equivalence classes of `P` cannot be split anymore: `P` is the Myhill-Nerode equivalence relation.

Avoiding automata isomorphism. The languages denoted by two regular expressions are equal if and only if their respective minimised automata are equal up-to isomorphism. By exploring all state permutations, this is sufficient to obtain decidability of regular languages equality. One can do a little better, however: it is not necessary to look for such a permutation. Suppose that languages α and β are represented by two DFAs; minimise the automaton whose set of states is the union of the states of the DFAs (i.e., the sum automaton), and test if the initial states of the two original DFAs are merged: these states are equivalent if and only if the DFAs recognise the same language, i.e., $\alpha = \beta$. This ends our description of the algorithm.

5 Conclusions and directions for future work

We presented a 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. Although we still have to finish some proofs (essentially in the analysis of the determinisation and minimisation algorithms), this tactic and the related tools can already be used; the development can be downloaded from [6]. To our knowledge, this is the first efficient implementation of these algorithms in Coq, and their integration into a generic tactic. At the time we started this project, Briais formalised decidability of regular languages equalities [7] (but not Kozen's initiality theorem), without taking care about efficiency: determinisation is always exponential; instead of minimising automata, he relies on the 'pumping lemma' to enumerate the finite set of accepted 'small enough' words. As a consequence, even straightforward identities cannot be checked by letting Coq compute. These preliminary results lead us to restart from scratch and to look for a better strategy.

We conclude this paper with directions for future work.

Optimisations. Even if this tactic works almost instantaneously on simple examples, such as the ones appearing in typical algebraic proofs, there is room for optimisation.

- We use unary integers to represent states; this is a drawback when we memoise matrices or make comparisons of state sets. A first step would be to move to Coq's binary natural numbers (`N`); we plan to resort eventually to either n-ary integers [13], or machine integers [25].
- Although the algorithms we implemented for determinisation and minimisation are optimal, this is not the case for Thomson's construction: we could use other algorithms [3, 8], that produce smaller automata. The complexity of these algorithms is slightly greater (typically $O(n^2)$ while Thomson's construction works in $O(n)$), but this has to be neglected w.r.t. the cost of determinisation, which is potentially exponential in the size of the starting NFA. Therefore, being able to produce smaller automata should improve the overall complexity. These algorithms are more involved than Thomson's one, so that formalising their correctness should not be straightforward.

Richer algebras. Kleene algebras lack several important operations from binary relations: intersection, converse, complement, residuals... We would like to develop tools for the corresponding algebras:

- *Kleene algebras with converse* should be decidable: since the converse operation commutes with all operations, we can imagine to push converses to the leafs of the terms, before applying our tactic for Kleene algebras.
- *Residuated semirings* [16], i.e., semirings with residual operations are decidable thanks to a Gentzen proof system having the sub-formula property. We plan to implement proof search for this proof system, either directly in `Ltac`, or using an external program to produce a trace that would then be reinterpreted as a Coq proof.
- *Allegories* [10] or *relation algebras* have an undecidable equational theory; they however provide means of encoding properties like well-foundedness [9], so that it would be interesting to provide tools for these structures (e.g., for solving decidable fragments).

Rewriting modulo A/AC. As explained in the introduction, some technology is required in order to work implicitly modulo associativity (A) and/or commutativity (C). For example, in the contexts below, we would like to rewrite the goal using hypothesis H without having to manually rearrange the goal first.

```
R,S,U,V: X            R,S,U,V: X          R,S,U,V: X
H: U·V = R            H: U+V = R          H: ∀ T, T·(T+U+V) = T
===========           ==========          =====================
(U·U)·V = S           V+S+U = S           (U·R)·(V+R+U) = S
```

For this development, we wrote ad-hoc tactics `ac_rewrite` and `monoid_rewrite` that work in simple cases like the first two examples. However, a more systematic approach is required in order to handle situations like the third one. We plan to pursue Beauquier's work on this topic [2]: we would like to implement algorithms for matching modulo A and AC [5], and to integrate the resulting (external) program with Coq, in order to obtain more satisfying tools for rewriting modulo A and AC.

References

[1] S. F. Allen, R. L. Constable, D. J. Howe, and W. E. Aitken. The semantics of reflected proof. In *Proc. LICS '90*, pages 95–105. IEEE Computer Society, 1990.

[2] M. Beauquier. Application du filtrage modulo associativité et commutativité (AC) à la réécriture de sous-termes modulo AC dans Coq. Master's thesis, Master Parisien de Recherche en Informatique, 2008.

[3] G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48(3):117–126, 1986.

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

[5] A. Boudet, E. Contejean, and H. Devie. A new AC unification algorithm with an algorithm for solving systems of diophantine equation. In *Proc. LICS '90*, pages 289–299. IEEE Computer Society Press, 1990.

[6] T. Braibant and D. Pous. Coq development: Algebraic tactics for working with binary relations. Avalaible from `http://sardes.inrialpes.fr/~braibant/atwbr/`, May 2009.

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

[8] J.-M. Champarnaud and D. Ziadi. Computing the equation automaton of a regular expression in space and time. In *Proc. CPM*, pages 157–168, 2001.

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

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

[11] F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging mathematical structures. In *Proc. TPHOL '09 (to appear)*, 2009. Available as a report from `http://hal.inria.fr/inria-00368403/`.

[12] B. Grégoire and A. Mahboubi. Proving equalities in a commutative ring done right in Coq. In *Proc. TPHOL '05*, LNCS, pages 98–113. Springer Verlag, 2005.

[13] B. Grégoire and L. Théry. A purely functional library for modular arithmetic and its application for certifying large prime numbers. In *Proc. IJCAR '06*, volume 4130 of *LNAI*, pages 423–437. Springer Verlag, 2006.

[14] D. Gries. Describing an algorithm by Hopcroft. *Acta Informatica*, 2:97–109, 1973.

[15] J. E. Hopcroft. An $n\log n$ algorithm for minimizing states in a finite automaton. Technical report, Stanford University, 1971.

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

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

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

[19] D. Kozen. Typed Kleene algebra. Technical Report TR98-1669, Computer Science Department, Cornell University, March 1998.

[20] A.R. Meyer and L. J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *Proc. SWAT '72*, pages 125–129. IEEE Computer Society, 1972.

[21] A.R. Meyer and L. J. Stockmeyer. Word problems requiring exponential time. In *Proc. STOC '73*, pages 1–9. ACM, 1973.

[22] A. Nerode. Linear automaton transformations. In *Proc. of the AMS*, volume 9, pages 541–544, 1958.

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

[24] M. Sozeau and N. Oury. First-class type classes. In *TPHOLs*, pages 278–293, 2008.

[25] A. Spiwack. Ajouter des entiers machine à Coq. `http://arnaud.spiwack.free.fr/papers/nativint.pdf`, 2006.

[26] K. Thompson. Regular expression search algorithm. *Comm. of the ACM*, 11:419–422, 1968.

[27] B. W. Watson. A taxonomy of finite automata construction algorithms. Technical report, Computing Science, 1993.