An Efficient Coq Tactic for Deciding Kleene Algebras

Thomas Braibant, Damien Pous

An Efficient Coq Tactic for Deciding Kleene Algebras*

Thomas Braibant and Damien Pous

LIG, UMR 5217, CNRS – INRIA

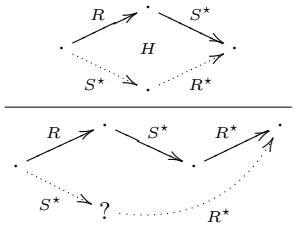
Abstract. We present a reflexive tactic for deciding the equational theory of Kleene algebras in the Coq proof assistant. This tactic relies on a careful implementation of efficient finite automata algorithms, so that it solves casual equations almost instantaneously. The corresponding decision procedure was proved correct and complete; correctness is established w.r.t. any model (including binary relations), by formalising Kozen's initiality theorem.

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 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.

* To appear in Proc. ITP, vol. 6172 of LNCS, July, 2010. A preliminary version of this work was presented at the 1st Coq Workshop, 2009.




```
R,S: relation P
H: ∀p,r,q, R p r → S⋆ r q
    → ∃s, S⋆ p s & R⋆ s q
p,q,q',s: P
Hpq: R p q
Hqs: S⋆ q s
Hsq': R⋆ s q'
=========================
∃s: P, S⋆ p s & R⋆ s q'
```

```
R,S: X
H: R·S⋆ ⊆ S⋆·R⋆
===============
R·S⋆·R⋆ ⊆ S⋆·R⋆
```

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

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, once we rewrite H in the right-hand side goal of Fig. 1, we obtain the inclusion $S^\star \cdot R^\star \cdot R^\star \subseteq S^\star \cdot R^\star$ which is a (straightforward) theorem of Kleene algebras: the tactic we describe in this paper proves it automatically.

Outline. We give some mathematical background and we sketch the overall structure of the tactic in Sect. 1. The underlying design choices are discussed in Sect. 2. We describe the algorithm and its correctness proof in Sect. 3. Section 4 is devoted to related works and directions for future work.

1 Deciding equalities in Kleene algebras

Theoretical background. A Kleene algebra [20] 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 $x \leq y \triangleq x + y = y$):

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

Terms of Kleene algebras are called *regular expressions*, irrespective of the considered model. 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. Thanks to finite automata theory [19,29], 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 decide equations in all Kleene algebras: it only applies to the regular languages model. We actually need a more recent theorem, by Kozen [20] (independently proved by Krob [24]):

> *"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 finite automata algorithms to solve equations in an arbitrary Kleene algebra $\mathcal{A}$. The main idea of Kozen's proof is to encode finite automata using matrices over $\mathcal{A}$, and to replay the algorithms at this algebraic level. Indeed, a finite automaton with transitions labelled by the elements of $\mathcal{A}$ 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.

We 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. Therefore, to mimic the behaviour of a finite automaton, 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: *"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 $u \cdot M^\star \cdot v$.

Overview of our strategy. We define a *reflexive* tactic. This methodology is quite standard [2]: for example, this is how the `ring` tactic is implemented [14]. Concretely, this means that we implement the decision procedure as a Coq program, and that we prove its correctness and completeness within the proof assistant:

```
Definition decide_kleene: regex → regex → bool := ...
Theorem Kozen: ∀ a b, decide_kleene a b = true ↔ a == b.
```

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 given set of variables, and `==` is the inductive equality generated by the axioms of Kleene algebras and the rules of equational reasoning (see Fig. 4). Using reification mechanisms, this is sufficient for our needs: the result can be lifted to other models using simple tactics (see Sect. 2.3).

The equational theory of Kleene algebras is PSPACE-complete [26]; this means that the `decide_kleene` function must be written with care, using efficient out-of-the-shelf algorithms. Notably, the matricial representation of automata is not efficient, so that formalising Kozen's "mathematical" proof [20] in a naive way would be computationally impracticable. Instead, we need to chose appropriate data structures for automata and algorithms, and to rely on the matricial representation only in proofs, using the adequate translation functions.

2 Underlying design choices

Before giving more details about our implementation of the decision procedure (Sect. 3), we explain the main choices we made about the design of our library: definition of the algebraic hierarchy, representation of matrices and handling of heterogeneous structures, reification mechanism, and numbers representation.

2.1 Algebraic hierarchy

The mathematical definition of a Kleene algebra is incremental: a Kleene algebra is a non-commutative semiring, which is itself composed of a monoid and a semilattice. Moreover, proofs naturally follow this hierarchy: when proving results about semirings, one usually relies on results about both monoids and semilattices. In order to structure our development in a similar way, we defined the algebraic hierarchy using Coq's recent *typeclasses* mechanism [30]: we defined several classes, corresponding to the different algebraic structures, so as to obtain the following "sub-class" relations:

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

Records vs. modules. Typeclasses are based on *records*; another possibility was to use *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 than modules for defining such a hierarchy. 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 there are nice and simple solutions for overloading notations (e.g., we can use the same infix symbol for multiplication in a monoid, a semiring, or a matrix semiring).

Typeclasses vs. canonical structures. *Canonical structures* is another record-based inference mechanism (which we incidentally use to declare concrete structures). We tried to use canonical structures instead of typeclasses to define the algebraic hierarchy from the beginning, along the lines of [13,3,12]. The overall benefit was unclear. This is mainly because our hierarchy is not really deep, so that we can handle it with typeclasses without introducing a complex infrastructure (we reached the limit, however: some of our proofs are noticeably slow to compile, due to this simplistic approach). Another reason is that we lack a deep understanding of Coq internal unification algorithm, which currently seems to be required to work efficiently with canonical structures. Therefore, we might switch to canonical structures at some point, when this technology will be better understood and supported.

2.2 Matrices

A matrix can be seen as a partial map from pairs of integers to a given type X, so that a Coq type for matrices and a sum operation could be defined as follows:

```
Definition MX (n m: nat) := ∀ i j, i<n → j<m → X.
Definition 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.
```

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 [3,12,4]. 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 above `plus` function; this however requires more boilerplate for other functions, like block decomposition operations.

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 equal n m (M N: MX n m) := ∀ i j, i<n → j<m → M i j == N i j.
Fixpoint sum i k (f: nat → X) :=
    match k with 0 ⇒ 0 | S k ⇒ f i + sum (S i) k f end.
Definition dot n m p (M: MX n m) (N: 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 subsequent definitions are easier to write. Bounds checks are required a posteriori only, when proving properties about these matrices operations, e.g., associativity of the product. This is generally straightforward: these proofs are done within the interactive proof mode, so that they can be solved with high level tactics like `omega`. (Note that this separation between proofs and programs could also be achieved syntactically—even with a dependently typed definition of matrices—by using Coq's `Program` feature.)

Although the correctness proof of our algorithm heavily relies on matricial reasoning, and in particular block matrix decompositions, 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, e.g., linear algebra [12].

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 because of the conversion rule: since the dependent type `MX n m` does not mention `n` nor `m` in its body, these type informations can be discarded by the type system using the conversion rule (`MX n 16 = MX n 64`). While such an ill-formed definition will be detected at proof-time; it is a bit sad not to benefit from 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: nat) (M: MX n m) := match M 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 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`).

```
X: Type.

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

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

```
T: Type.
X: T → T → Type.

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

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

Fig. 2. From Kleene algebras to typed Kleene algebras.

Computation. Although we do not use matrices in 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 (e.g., matrix multiplications). Second, using functions to represent matrices is two-edged: on the one hand, if the matrix resulting of a computation is seldom used, then computing its elements by need is efficient; on the other hand, making numerous accesses to the same expensive computation may be a burden. To this end, we defined 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. This memoisation operator is proved to be an identity; 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_lists M in box n m (fun i j ⇒ nth i (nth j l)).
Lemma mx_force_id : ∀ n m (M : MX n m), mx_force M == M.
```

2.3 Typed algebras, typed reification

Adding types. We need to work with *rectangular* matrices at several places: to prove the correctness of some steps of the algorithm (see Sect. 3.3), and to treat vectors as matrices so as to factorise proofs. However, while *square* matrices over a semiring form a semiring, this is not the case 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 indices (or types) is used to constrain the various operations. These abstract 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.

As expected, with such definitions, one can form arbitrary matrices over a typed structure, and obtain another instance of this typed structure. In particular, the matrices over an arbitrary typed Kleene algebra form a typed Kleene

algebra. 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. This is quite important since this construction is rather complicated.

Removing types. Typed structures not only make it easier to work with rectangular matrices, they also give rise to a wider range of models. In particular, we can consider heterogeneous binary relations rather than binary relations on a single fixed set. This leads to the following question: can the usual decision procedures 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 $n \to m$ and $m \to n$, for some n, m. How to ensure that the untyped automata algorithms respect types and actually give valid, well-typed, proofs?

For efficiency and practicability reasons, extending our decision procedure to work with typed elements is not an option. Instead, we proved the following theorem, which allows one to erase types, i.e., to transform a typed equality goal into an untyped one:

$$\frac{\vdash u = v \qquad \Gamma \vdash u \rhd \alpha : n \to m \qquad \Gamma \vdash v \rhd \beta : n \to m}{\vdash \alpha = \beta : n \to m} \qquad (*)$$

Here, $\Gamma \vdash u \rhd \alpha : n \to m$ reads "under the evaluation and typing context Γ, the untyped term u can be evaluated to α, of type $n \to m$"; this predicate can be defined inductively in a straightforward way, for various algebraic structures. The theorem can then be rephrased as follows: "if the untyped terms u and v are equal, then for all typed interpretations α and β of u and v, the typed equality $\alpha = \beta$ holds". See [28] for a theoretical study of these untyping theorems; also note that Kozen investigated a similar question [21] 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.

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` mechanism, which works by conversion and has no impact on the size of proofs, this simple "user-level"-`quote` generates

large proof-terms. In fact, this is the current bottleneck of our tactic: on casual examples, the time spent in reification exceeds the time spent in computations! We thus plan to implement an efficient reification mechanism, possibly in OCaml.

2.4 Numbers, finite sets, and finite maps

To code our decision procedure, we mainly need 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. 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's 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`, that 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.

3 The algorithm and its proof

We now focus on the heart of our tactic: the decision procedure and the corresponding correctness proof. The algorithm that decides whether two regular expressions denote the same language can be decomposed into four steps:

1. build non-deterministic finite automata with epsilon-transitions (ϵ-NFA);
2. remove epsilon-transitions to get non-deterministic finite automata (NFA);
3. determinise the automata to obtain deterministic finite automata (DFA);
4. check that the two DFAs are equivalent.

The third 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. 3; we use modules only for handling the namespace; the type `regex` is defined in Fig. 4, `label` and `state` are synonyms 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; we use it only in proofs, through the evaluation function `MAUT.eval` (`MX n m` is the type of $n \times m$ matrices over `regex`; the evaluation function calculates the matricial product $u \cdot M^\star \cdot v$ and casts it to

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

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

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

Fig. 3. Coq types and evaluation functions of the four automata representations.

a `regex`). 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. 5. Datatypes are recalled on the left-hand side; the outer part of the right-hand side corresponds to computations: starting from two regular expressions α and β, two DFAs A_3 and B_3 are constructed 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 do not implement the same algorithms, this proof is largely based on Kozen's one [20]).

3.1 Construction

There are several ways of constructing an ϵ-NFA from a regular expression. At first, we implemented Thompson's construction [33], for its simplicity; we finally switched to a variant of the Ilie and Yu's construction [17] 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

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

Inductive eq :=
| eq_trans: Transitive eq
| eq_sym: Symmetric eq
| eq_dot_zero: ∀e, e·0==0
| eq_plus_idem: ∀e, e+e==e
| ...
where "e==f" := (eq e f).
```

Fig. 4. Regular expressions.

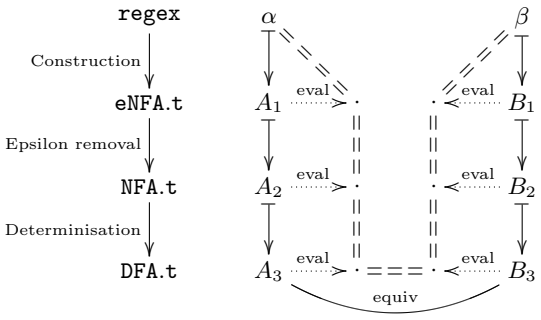
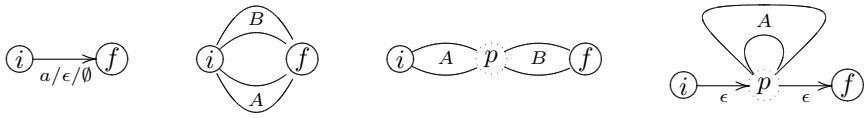


Fig. 5. Correctness of the algorithm.

depicted below; the first drawing corresponds to the base cases (variable, one, zero); 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.



To avoid costly union operations, we actually use an accumulator (an ϵ-NFA) to which we recursively add states and transitions.

We prove correctness in two steps, by using a more high-level algorithm. This algorithm is very similar to the previous one, except that it works with the matricial representation (`MAUT.t`). The following lemma states that the corresponding implementations are equivalent (`regex_to_eNFA` is the efficient function, `regex_to_MAUT` is the high-level one, and `===` is matricial automata equality):

```
Lemma step1: ∀e, eNFA.to_MAUT (regex_to_eNFA e) === regex_to_MAUT e.
```

Therefore, it suffices to prove correctness for the high-level construction—the following lemma—for which we can use algebraic and matricial reasoning.

```
Lemma step2: ∀e, MAUT.eval (regex_to_MAUT e) == e.
```

To obtain the latter lemma, we have to consider the following one, where `build` is the recursive function that underpins `regex_to_MAUT`: `build e i f A` applies the above construction to the regular expression `e`, between states `i` and `f` of the matricial automaton accumulator `A`; and `add e i f A` just adds a transition labelled `e` to `A`, between `i` and `f`.

```
Lemma step2': ∀e i f A, MAUT.eval (build e i f A) == MAUT.eval (add e i f A).
```

As an example of the kind of algebraic reasoning we need to formalise, the following property of star w.r.t. block matrices is used twice: 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 $(e^\star)$. 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.

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

Finally, by combining the lemmas `step1` and `step2`, we obtain the correctness of our construction algorithm, i.e., we can fill the two triangles from Fig. 5:

```
Theorem construction_correct: ∀e, eNFA.eval (regex_to_eNFA e) == e.
```

3.2 Epsilon transitions removal

The automata obtained with the above construction contain epsilon-transitions: 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. J and N respectively correspond to the graph of epsilon and labelled transitions. Algebraically, removing epsilon-transitions is achieved using a simple law: $\forall xy, (x + y)^\star = x^\star \cdot (y \cdot x^\star)^\star$. This law yields

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

so that automata $\langle u, N, v\rangle$ and $\langle u \cdot J^\star, N \cdot J^\star, v\rangle$ are equivalent. We furthermore check that the latter automaton no longer contains epsilon-transitions: this is a NFA. This algebraic proof is not surprising: looking at 0-1 matrices as binary relations, $J^\star$ 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. 3.1: when given regular expressions in "strict star" form, the produced ϵ-NFAs have acyclic epsilon-transitions. More precisely, we say that a regular expression is in *strict star form* if for all its sub-expressions of the form $e^\star$, the regular language corresponding to e does not contain the empty word. Intuitively, the only possibility for introducing an epsilon-cycle in the construction from Sect. 3.1 comes from star expressions. By forbidding the empty word to appear in such cases, we prevent the formation of epsilon-cycles.

Concretely, this means that: 1) we wrote a recursive function that transforms a regular expression into an equivalent one, in strict star form; 2) we proved that our construction algorithm returns ϵ-NFAs whose reversed epsilon-transitions are well-founded, when the argument is in strict star form; 3) based on this assumption we implemented a simple transitive closure algorithm, using well-founded recursion and memoisation; 4) we proved that this algorithm actually yields an automaton (of type `NFA.t`) 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.

3.3 Determinisation

Determinisation is exponential in worst case: this is a power-set construction. However, examples where this bound is reached are rather contrived: the empirical complexity is tractable. Starting from a NFA $\langle u, M, v\rangle$ with n states, the algorithm consists in a standard depth-first enumeration of the subsets 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 $X_{sj} \triangleq j \in \rho(s)$, we prove that the following commutation properties hold:

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

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. Therefore, (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. From (1), we deduce that $(\widehat{M})^\star \cdot X = X \cdot M^\star$ using a theorem of Kleene algebras; we conclude with $(2, 3)$:

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

A Coq-specific technical difficulty in the concrete implementation of this algorithm comes from termination: the main loop is executed at most 2^n times (there are 2^n subsets of $[1..n]$), but we cannot use this bound directly. Indeed, we can easily determinise NFAs with 500 states in practice, while computing 2^{500} is obviously out of reach (the binary representation of numbers does not help since we need to do structural "unary" recursion); we thus have to iterate lazily. We tried to use well-founded recursion; this was rather inconvenient, however, since 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 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.

3.4 Equivalence checking

Two DFAs are equivalent if and only if their respective minimised DFAs are equal up-to isomorphism. While exploring all state permutations is sufficient to obtain decidability, there is a more direct and efficient approach that does not require minimisation: one can perform an on-the-fly *simulation* check, using an almost linear algorithm by Hopcroft and Karp [1].

This algorithm proceeds as follow: it computes the disjoint union automata $\langle u, M, v\rangle$, and checks that the former initial states (i_A, i_B) are equivalent. 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. Hence, the algorithm starts with a pair of states that must be equivalent—typically (i_A, i_B)—and try to recursively equate their reducts along transitions. To be almost linear, the algorithm uses a disjoint-sets data structure to compute equivalence classes. Indeed, if the pairs $\{i, j\}$ and $\{j, k\}$ have already been equated, one can skip the pair $\{i, k\}$ if encountered.

To our knowledge, there are two implementations of union-find data structures in Coq [9,25]. However, [9] is not computational, and [25] is more geared toward extraction (it uses `sig` types). Therefore, we had to re-implement and prove this data structure from scratch. Namely, we implemented disjoint-sets forests [10] with path compression and the usual "union by rank" heuristic.

Like previously, the correctness of the equivalence check is proved algebraically: we define a 0-1 matrix Y to encode the obtained equivalence relation on states, and we prove that it satisfies the following properties:

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

$$i_A \cdot Y = i_B \cdot Y \quad (4) \qquad Y \cdot v = v \quad (5)$$

Equations $(1, 2)$ correspond to the fact that Y encodes a reflexive and transitive relation. The remaining equations assess that Y is a simulation (3), that the initial arguments are equated (4), and that related states are either accepting or non accepting (5). This allows us to conclude using standard algebraic reasoning.

3.5 Completeness: counter-examples

By combining the proofs from the above sections according to Fig. 5, we obtain the correctness of the decision procedure: if `decide_kleene a b` returns `true`, then `a==b`, and thanks to the untyping theorem $(*)$ from Sect. 2.3, we deduce that `a` and `b` are equal in any typed Kleene algebra.

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, this is almost straightforward: it suffices to record the word that is being read in the algorithm from Sect. 3.4; 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.

From the proof point of view, to obtain the reverse implication of the equivalence we mentioned in Sect. 1, we just have to show that languages (i.e., predicates over list of labels, `list label→Prop`) form a Kleene algebra in which the language accepted by a DFA is exactly the language obtained with `DFA.eval`:

```
Theorem interp_DFA_eval: ∀ A: DFA.t, DFA_language A [=] interp (DFA.eval A).
```

(`DFA.eval` actually returns a regular expression which we need to interpret as a language; `[=]` is language equality, i.e., pointwise equivalence of the predicates.)

4 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 can be downloaded from [5]. To our knowledge, this is the first certified efficient implementation of these algorithms and their integration as a generic tactic.

4.1 Performances

We performed intensive tests on randomly generated regular expressions. On typical use cases, the tactic runs instantaneously (except for the time spent in the reification mechanism, as explained at the end of Sect. 2.3). It runs in less than one second for expressions with 150 internal nodes and 20 variables, and less than one minute for even larger expressions (1000 internal nodes, 40 variables), that are very unlikely to appear in "human-written" proofs.

Thanks to the efficient implementation of radix-2 search trees (`PositiveMap`), we get higher performances if we use `positive` rather than `BigN.t`, despite the underlying mechanism that uses machine arithmetic—a recent feature of Coq.

4.2 Related works

The idea of reasoning about binary relations algebraically is old [32,11]. Among others [18,34], Struth applied this idea within an interactive theorem prover [31]. He later turned to automated first-order theorem provers (ATP): Höfner and him verified facts about various relation algebras [15,16] 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 [16].

At the time we started this project, Briais formalised decidability of regular languages equality [6]. However, his approach is not computational, so that even straightforward identities cannot be checked by letting Coq compute.

Narboux defined a set of Coq tactics for diagrammatic proofs [27]. 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.

Our notion of strict star form (Sect. 3.2) was inspired by the standard notion of *star normal form* [7] and the idea of *star unavoidability* [17]. To our knowledge, the facts that we can always rewrite regular expressions in such a form and that ϵ-NFAs with acyclic epsilon-transitions can be constructed in this way are new.

4.3 Directions for future work

Earlier failure checks. Our algorithm for checking equivalence of DFAs returns whenever two non-equivalent states are encountered. This optimisation greatly improves over minimisation-based algorithms, but we could go one step further, by checking the equivalence on-the-fly, during the determinisation phase. This way, we could abort the computation of DFAs as soon as a discrepancy is found.

KAT, Hoare logic. We plan to extend our decision procedure to *Kleene algebras with tests* (KAT), so as to provide automation to prove correctness of programs in Hoare logic [22]. A first possibility would be to encode KAT expressions into KA [23] and to use the current tactic. This encoding being potentially exponential, it is unclear whether this approach would be tractable. A more involved approach would be to use the dedicated automata construction presented in [8].

Acknowledgements. We thank Guilhem Moulin, Assia Mahboubi, Matthieu Sozeau, Bruno Barras, and Hugo Herbelin for highly stimulating discussions.

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. Y. Bertot, G. Gonthier, S. Ould Biha, and I. Pasca. Canonical big operators. In *TPHOL*, volume 5170 of *LNCS*, pages 86–101. Springer, 2008.
4. F. Blanqui, S. Coupet-Grimal, W. Delobel, and A. Koprowski. CoLoR: a Coq library on rewriting and termination, 2006.
5. T. Braibant and D. Pous. Coq library: ATBR, algebraic tools for working with binary relations. `http://sardes.inrialpes.fr/~braibant/atbr/`, May 2009.
6. S. Briais. Coq development: Finite automata theory. `http://www.prism.uvsq.fr/~bris/tools/Automata_080708.tar.gz`, July 2008.
7. A. Brüggemann-Klein. Regular expressions into finite automata. *TCS*, 120(2):197–213, 1993.

8. E. Cohen, D. Kozen, and F. Smith. The complexity of Kleene algebra with tests, July 1996. TR96-1598, CS Dpt., Cornell University.
9. S. Conchon and J.-C. Filliâtre. A Persistent Union-Find Data Structure. In *ACM SIGPLAN Workshop on ML*, pages 37–45, Freiburg, Germany, October 2007.
10. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, second edition, 2001.
11. H. Doornbos, R. Backhouse, and J. van der Woude. A calculational approach to mathematical induction. *TCS*, 179(1-2):103–135, 1997.
12. F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging mathematical structures. In *TPHOL*, volume 5674 of *LNCS*, pages 327–342. Springer, 2009.
13. G. Gonthier, A. Mahboubi, L. Rideau, E. Tassi, and L. Théry. A modular formalisation of finite group theory. In *TPHOL*, volume 4732 of *LNCS*, pages 86–101. Springer, 2007.
14. B. Grégoire and A. Mahboubi. Proving equalities in a commutative ring done right in Coq. In *TPHOL*, volume 3603 of *LNCS*, pages 98–113. Springer, 2005.
15. P. Höfner and G. Struth. Automated reasoning in Kleene algebra. In *CADE*, volume 4603 of *LNCS*, pages 279–294. Springer, 2007.
16. P. Höfner and G. Struth. On automating the calculus of relations. In *IJCAR*, volume 5195 of *LNCS*, pages 50–66. Springer, 2008.
17. L. Ilie and S. Yu. Follow automata. *Inf. and Comp.*, 186(1):140–162, 2003.
18. W. Kahl. Calculational relation-algebraic proofs in Isabelle/Isar. In *RelMiCS*, volume 3051 of *LNCS*, pages 178–190. Springer, 2003.
19. S. C. Kleene. Representation of events in nerve nets and finite automata. In *Automata Studies*, pages 3–41. Princeton University Press, 1956.
20. D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Inf. and Comp.*, 110(2):366–390, 1994.
21. D. Kozen. Typed Kleene algebra, 1998. TR98-1669, CS Dpt. Cornell University.
22. D. Kozen. On Hoare logic and Kleene algebra with tests. *ACM Trans. Comput. Log.*, 1(1):60–76, July 2000.
23. D. Kozen and F. Smith. Kleene algebra with tests: Completeness and decidability. In *CSL*, volume 1258 of *LNCS*, pages 244–259. Springer, September 1996.
24. D. Krob. Complete systems of B-rational identities. *TCS*, 89(2):207–343, 1991.
25. X. Leroy. A formally verified compiler back-end. *JAR*, 43(4):363–446, 2009.
26. A.R. Meyer and L. J. Stockmeyer. Word problems requiring exponential time. In *Proc. STOC*, pages 1–9. ACM, 1973.
27. J. Narboux. *Formalisation et automatisation du raisonnement géométrique en Coq*. PhD thesis, Université Paris Sud, September 2006.
28. D. Pous. Untyping typed algebraic structures and colouring proof nets of cyclic linear logic. Technical Report RR-7176, INRIA Rhône-Alpes, January 2010.
29. M.O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959.
30. M. Sozeau and N. Oury. First-class type classes. In *TPHOL*, volume 4732 of *LNCS*, pages 278–293. Springer, 2008.
31. G. Struth. Calculating Church-Rosser proofs in Kleene algebra. In *RelMiCS*, volume 2561 of *LNCS*, pages 276–290. Springer, 2001.
32. A. Tarski and S. Givant. *A Formalization of Set Theory without Variables*, volume 41 of *Colloquium Publications*. AMS, Providence, Rhode Island, 1987.
33. K. Thompson. Regular expression search algorithm. *C. ACM*, 11:419–422, 1968.
34. D. von Oheimb and T.F. Gritzner. RALL: Machine-supported proofs for relation algebra. In *CADE*, volume 1249 of *LNCS*, pages 380–394. Springer, 1997.