Kleene Algebra with Tests and Coq Tools for While Programs

Damien Pous

Kleene Algebra with Tests and Coq Tools for While Programs

Damien Pous

CNRS – LIP, ENS Lyon, UMR 5668

Abstract. We present a Coq library about Kleene algebra with tests, including a proof of their completeness over the appropriate notion of languages, a decision procedure for their equational theory, and tools for exploiting hypotheses of a certain kind in such a theory.

Kleene algebra with tests make it possible to represent if-then-else statements and while loops in most imperative programming languages. They were actually introduced by Kozen as an alternative to propositional Hoare logic.

We show how to exploit the corresponding Coq tools in the context of program verification by proving equivalences of while programs, correctness of some standard compiler optimisations, Hoare rules for partial correctness, and a particularly challenging equivalence of flowchart schemes.

Introduction

Kleene algebra with tests (KAT) have been introduced by Kozen [19], as an equational system for program verification. A Kleene algebra with tests is a Kleene algebra (KA) with an embedded Boolean algebra of tests. The Kleene algebra component deals with the control-flow graph of the programs—sequential composition, iteration, and branching—while the Boolean algebra component deals with the conditions appearing in if-then-else statements, while loops, or pre- and post-assertions.

This formalism is both concise and expressive, which allowed Kozen and others to give detailed paper proofs about various problems in program verification (see, e.g., [3, 19, 21, 23]). More importantly, the equational theory of KAT is decidable and complete over relational models [24], and hypotheses of a certain kind can moreover be eliminated [11, 15]. This suggests that a proof using KAT should not be done manually, but with the help of a computer. The goal of the present work is to give this possibility, inside the Coq proof assistant.

The underlying decision procedure cannot be formulated, a priori, as a simple rewriting system: it involves automata algorithms, it cannot be defined in Ltac, at the meta-level, and it does not produce a certificate which could easily be checked in Coq, a posteriori. This leaves us with only one possibility: defining a reflexive tactic [1, 8, 14]. Doing so is quite challenging: we basically have to prove completeness of KAT axioms w.r.t. the model of guarded string languages (the natural generalisation of languages for KA, to KAT), and to provide a provably correct algorithm for language equivalence of KAT expressions.

The completeness theorem is far from trivial; we actually have to formalise a lot of preliminary material: finite sums, finite sets, unique decomposition of Boolean expressions into sums of atoms, regular expression derivatives, expansion theorem for regular expressions, matrices, automata... As a consequence, we only give here a high-level overview of the involved mathematics, leaving aside standard definitions, technical details, or secondary formalisation tricks. The interested reader can consult the library, which is documented [30].

Outline. We first present KAT and its models (§1). We then sketch the completeness proof (§2), the decision procedure (§3), and the method used to eliminate hypotheses (§4). We finally illustrate the benefits of our tactics on several case-studies (§5), before discussing related works (§6), and concluding (§7).

1 Kleene Algebra with Tests

A Kleene algebra with tests consists of:

- a Kleene algebra $\langle X, \cdot, +, \cdot^\star, 1, 0\rangle$ [18], i.e., an idempotent semiring with a unary operation, called "Kleene star", satisfying an axiom: $1 + x \cdot x^\star \leq x^\star$ and two inference rules: $y \cdot x \leq x$ entails $y^\star \cdot x \leq x$ and the symmetric one. (The preorder $(\leq)$ being defined by $x \leq y \triangleq x + y = y$.)
- a Boolean algebra $\langle B, \wedge, \vee, \neg, \top, \bot\rangle$;
- a homomorphism from $\langle B, \wedge, \vee, \top, \bot\rangle$ to $\langle X, \cdot, +, 1, 0\rangle$, that is, a function $[\cdot] : B \to X$ such that $[a \wedge b] = [a] \cdot [b]$, $[a \vee b] = [a] + [b]$, $[\top] = 1$, and $[\bot] = 0$.

The elements of the set B are called "tests"; we denote them by a, b. The elements of X, called "Kleene elements", are denoted by x, y, z. We usually omit the operator "$\cdot$" from expressions, writing xy for $x \cdot y$. The following (in)equations illustrate the kind of laws that hold in all Kleene algebra with tests:

$$[a \vee \neg a] = 1 \qquad\qquad [a \wedge (\neg a \vee b)] = [a][b] = [\neg(\neg a \vee \neg b)]$$

$$x^\star x^\star = x^\star \qquad\qquad (x+y)^\star = x^\star(yx^\star)^\star \qquad\qquad (x+xxy)^\star \leq (x+xy)^\star$$

$$[a]([\neg a]x)^\star = [a] \qquad\qquad [a]([a]x[\neg a] + [\neg a]y[a])^\star[a] \leq (xy)^\star$$

The laws from the first line come from the Boolean algebra structure, while the ones from the second line come from the Kleene algebra structure. The two laws from the last line are more interesting: their proof must mix both Boolean algebra and Kleene algebra reasoning. They are left to the reader as a non-trivial exercice; the tools we present in this paper allow one to prove them automatically.

1.1 The model of binary relations

Binary relations form a Kleene algebra with tests; this is the main model we are interested in, in practice. The Kleene elements are the binary relations over a given set S, the tests are the predicates over this set, and the star of a relation is its reflexive transitive closure:

$$\begin{aligned}
X &= \mathcal{P}(S \times S) & B &= \mathcal{P}(S) \\
x \cdot y &= \{(p,q) \mid \exists r, (p,r) \in x \wedge (r,q) \in y\} & a \wedge b &= a \cap b \\
x + y &= x \cup y & a \vee b &= a \cup b \\
x^\star &= \{(p_0,p_n) \mid \exists p_1 \dots p_{n-1}, \forall i < n, (p_i, p_{i+1}) \in x\} & \neg a &= S \setminus a \\
1 &= \{(p,p) \mid p \in S\} & \top &= S \\
0 &= \emptyset \qquad\qquad [a] = \{(p,p) \mid p \in a\} & \bot &= \emptyset
\end{aligned}$$

The laws of a Kleene algebra are easily proved for these operations; note however that one needs either to restrict to decidable predicates (i.e., to take `S → bool` or `{p: S → Prop | forall p, S p ∨ ¬S p}` for B), or to assume the law of excluded middle: B must be a Boolean algebra, so that negation has to be an involution. This choice for B is left to the user of the library.

This relational model is typically used to interpret imperative programs: such programs are state transformers, i.e., binary relations between states, and the conditions appearing in these programs are just predicates on states. These conditions are usually decidable, so that the above constraint is actually natural.

The equational theory of Kleene algebra with tests is complete over the relational model [24]: any equation $x = y$ that holds universally in this model can be proved from the axioms of KAT. We do not need to formalise this theorem, but it is quite informative in practice: by contrapositive, if an equation cannot be proved from KAT, then it cannot be universally true on binary relations, meaning that proving its validity for a particular instantiation of the variables necessarily requires one to exploit additional properties of this particular instance.

1.2 Other models

We describe two other models in the sequel: the syntactic model (§1.3) and the model of guarded string languages (§1.4); these models have to be formalised to build the reflexive tactic we aim at.

There are other important models of KAT. First of all, any Kleene algebra can be extended into a Kleene algebra with tests by embedding the two-element Boolean lattice. We also have traces models (where one keeps track of the whole execution traces of the programs rather than just their starting and ending points), matrices over a Kleene algebra with tests, but also models inherited from semirings like min-plus and max-plus algebra. The latter models have a degenerate Kleene star operation; they become useful when one constructs matrices over them, for instance to study shortest path algorithms.

Also note that like for Kleene algebra [9, 20, 29], KAT admits a natural "typed" generalisation, allowing for instance to encompass heterogeneous binary relations and rectangular matrices. Our Coq library is actually based on this generalisation, and this deeply impacts the whole infrastructure; we however omit the corresponding details and technicalities here, for the sake of clarity.

1.3 KAT expressions

Let p, q range over a set Σ of *letters* (or *actions*), and let $a_1, \dots, a_n$ be the elements of a finite set Θ of *primitive tests. Boolean expressions* and *KAT expressions* are defined by the following syntax:

$$\begin{array}{lr} a, b ::= a_i \in \Theta \mid a \wedge a \mid a \vee a \mid \neg a \mid \top \mid \bot & \text{(Boolean expressions)} \\ x, y ::= p \in \Sigma \mid [a] \mid x \cdot y \mid x + y \mid x^\star \mid 1 \mid 0 \ . & \text{(KAT expressions)} \end{array}$$

Given a Kleene algebra with tests $\mathcal{K} = \langle X, B, [\cdot] \rangle$, any pair of maps $\theta : \Theta \to B$ and $\sigma : \Sigma \to X$ gives rise to a KAT homomorphism allowing to interpret expressions in $\mathcal{K}$. Given two such expressions x and y, the equation $x = y$ is a *KAT theorem*, written $\mathrm{KAT} \vdash x = y$, when the equation holds in any Kleene algebra with tests, under any interpretation. One checks easily that KAT expressions quotiented by the latter relation form a Kleene algebra with tests; this is the free Kleene algebra with tests over Σ and Θ. (We actually use this impredicative encoding of KAT derivability in the Coq library.)

1.4 Guarded strings languages

Guarded string languages are the natural generalisation of string languages for Kleene algebra with tests. We briefly define them.

An *atom* is a function from elementary tests (Θ) to Booleans; it indicates which of these tests are satisfied. We let α, β range over atoms, the set of which is denoted by At. (Technically, we represent elementary tests as finite ordinals of a given size n ($\Theta = \texttt{ord}\ n$), and we encode atoms as ordinals ($\texttt{At} = \texttt{ord}\ 2^n$). This allows us to avoid functional extensionality problems.) We let u, v range over *guarded strings*: alternating sequences of atoms and letters, which both start and end with an atom:

$$\alpha_1, p_1, \dots, \alpha_n, p_n, \alpha_{n+1} \ .$$

The concatenation $u * v$ of two guarded strings u, v is a partial operation: it is defined only if the last atom of u is equal to the first atom of v; it consists in concatenating the two sequences and removing the shared atom in the middle.

The Kleene algebra with tests of guarded string languages is obtained by considering sets of guarded strings for X and sets of atoms for B:

$$\begin{array}{rlr} X = \mathcal{P}\left((At \times \Sigma)^\star \times At\right) & & B = \mathcal{P}\left(At\right) \\ x \cdot y = \{u * v \mid u \in x \wedge v \in y\} & & a \wedge b = a \cap b \\ x + y = x \cup y & & a \vee b = a \cup b \\ x^\star = \{u_1 * \dots * u_n \mid \exists u_1 \dots u_n, \forall i \leq n, u_i \in x\} & & \neg a = At \setminus a \\ 1 = \{\alpha \mid \alpha \in At\} & & \top = At \\ 0 = \emptyset & [a] = \{\alpha \mid \alpha \in a\} & \bot = \emptyset \end{array}$$

Note that we slightly abuse notation by letting α denote either an atom, or a guarded string reduced to an atom. Also note that the set $B = \mathcal{P}\left(At\right)$ has to be represented by the Coq type $\texttt{At} \to \texttt{bool}$, to get a Boolean algebra on it.

2 Completeness

Let G be the unique homomorphism from KAT expressions to guarded string languages such that

$$G(a_i) = \{\alpha \mid \alpha(a_i) \text{ is true}\} \qquad\qquad G(p) = \{\alpha p \beta \mid \alpha, \beta \in At\}$$

Completeness of KAT over guarded string languages can be stated as follows.

Theorem 1. *For all KAT expressions x, y, $G(x) = G(y)$ entails* $\mathrm{KAT} \vdash x = y$.

This theorem is central to our development: it allows us to prove (in)equations in arbitrary models of KAT, by resorting to an algorithm deciding guarded string language equivalence (to be described in §3).

We closely follow Kozen and Smith' proof [24]. This proof relies on the completeness of Kleene algebra over languages, which we thus need to prove first.

2.1 Completeness of Kleene algebra axioms

Let R be the Kleene algebra homomorphism from regular expressions to (plain) string languages mapping a letter p to the language consisting of the single-letter word p. KA completeness over languages can be stated as follows [18]:

Theorem 2. *For all regular expressions x, y, $R(x) = R(y)$ entails* $\mathrm{KA} \vdash x = y$.

(Like for KAT, the judgement $\mathrm{KA} \vdash x = y$ means that $x = y$ holds in any Kleene algebra, under any interpretation.) We already presented a Coq formalisation of this theorem [9], but our development was over-complicated. We re-proved it from scratch here, following a simpler path which we now describe.

The main idea of Kozen's proof consists in replaying automata algorithms algebraically, using matrices to encode automata. The key insight that allowed us to considerably simplify the corresponding formalisation is that the algorithm used for this proof need not be the same as the one to be executed by the reflexive tactic we eventually define. Indeed, we can take the simplest possible algorithm to prove KA completeness, ignoring all complexity aspects, thus allowing us to focus on conciseness and mathematical simplicity. In contrast, the algorithm to be executed by the final reflexive tactic should be relatively efficient, but we do not need to prove it complete, nor to replay its correctness algebraically: we only need to prove its correctness w.r.t. languages, which is much easier.

A preliminary step for the proof consists in proving that matrices over a Kleene algebra form a Kleene algebra. The Kleene star for matrices is non-trivial to define and to prove correct, but this can be done with a reasonable amount of efforts once appropriate lemmas and tools for block matrices have been set up.

A finite automaton can then be represented using three matrices (u, M, v) over regular expressions, where u is a $(1, n)$-matrix, M is a (n, n)-matrix, and v is a $(n, 1)$-matrix, n being the number of states of the automaton. Such a "matricial automaton" can be evaluated into a regular expression by taking the product $u \cdot M^\star \cdot v$, which is a scalar. The various classes of automata can be recovered by imposing conditions on the coefficients of the three matrices. For instance, a non-deterministic finite automaton (NFA) is such that u and v are 01-vectors and the coefficients of M are sums of letters.

Given a regular expression x, we construct a deterministic finite automaton (DFA) (u, M, v) such that $\mathrm{KA} \vdash x = uM^\star v$, as follows.

1. First construct a NFA with epsilon transitions (u'', M'', v''), such that $\mathrm{KA} \vdash x = u''M''^\star v''$. This is easily done by induction on x, using Thompson construction [31] (which is compositional, unlike the construction we used in [9]).
2. Remove epsilon transitions to obtain a NFA (u', M', v') such that $\mathrm{KA} \vdash u''M''^\star v'' = u'M'^\star v'$. We do it purely algebraically, in one line. In particular the transitive closure of epsilon transitions is computed using Kleene star on matrices. (Unlike in [9] we do not need a dedicated algorithm for this.)
3. Use the powerset construction to convert this NFA into a DFA (u, M, v) such that $\mathrm{KA} \vdash u'M'^\star v' = uM^\star v$. Again, this is done algebraically, and we do not need to perform the standard 'accessible subsets' optimisation.

We can prove that for any DFA (u, M, v), $R(uM^\star v)$ is the language recognised by the DFA. Therefore, to obtain Theorem 2, it suffices to prove that if two DFA (u, M, v) and (s, N, t) recognise the same language, then $\mathrm{KA} \vdash uM^\star v = sN^\star t$. For this last step, it suffices to exhibit a Boolean matrix that relates exactly those states of the two DFA that recognise the same language. We need for that an algorithm to check language equivalence of DFA states; we reduce the problem to DFA emptiness, and we perform a simple reachability analysis.

All in all, the KA completeness proof itself only requires us 124 lines of specifications, and 119 lines of proofs (according to `coqwc`).

2.2 Completeness of KAT axioms

To obtain KAT completeness (Theorem 1), Kozen and Smith [24] define a function $\widehat{\cdot}$ on KAT expressions that expands the expressions in such a way that we have $\mathrm{KAT} \vdash x = y$ iff $\mathrm{KA} \vdash \widehat{x} = \widehat{y}$. While this function can be thought as a reduction of KAT to KA, it cannot be used in practice: it produces expressions that are almost systematically exponentially larger than the given ones. It is however sufficient to establish completeness; as explained earlier, we defer actual computations to a completely different algorithm (§3).

More precisely, the function $\widehat{\cdot}$ is defined in such a way that we have:

$$\mathrm{KAT} \vdash \widehat{x} = x \tag{i}$$

$$G(\widehat{x}) = R(\widehat{x}) \tag{ii}$$

We deduce KAT completeness as follows:

$$
\begin{aligned}
& G(x) = G(y) && \\
\Leftrightarrow\ & G(\widehat{x}) = G(\widehat{y}) && (G \text{ is a KAT morphism, and (i)}) \\
\Leftrightarrow\ & R(\widehat{x}) = R(\widehat{y}) && (\text{by (ii)}) \\
\Rightarrow\ & \mathrm{KA} \vdash \widehat{x} = \widehat{y} && (\text{KA completeness}) \\
\Rightarrow\ & \mathrm{KAT} \vdash \widehat{x} = \widehat{y} && (\text{any KAT is a KA}) \\
\Leftrightarrow\ & \mathrm{KAT} \vdash x = y && (\text{by (i)})
\end{aligned}
$$

(Note that the last equation entails the first one, so that all these statements are in fact equivalent.)

The function $\widehat{\cdot}$ is defined recursively over KAT expressions, using an intermediate datastructure: formal sums of *externally guarded terms* (i.e., either an atom, or a product of the form $\alpha x \beta$). The case of a starred expression $x^\star$ is quite involved: $\widehat{x^\star}$ is defined by an internal recursion on the length of the formal sum corresponding to $\widehat{x}$. The proof of the first equation (i) is not too difficult to formalise, using appropriate tools for finite sums (i.e., a simplified form of big operators [7], which we actually use a lot in the whole development). The second one (ii) is more cumbersome, notably because we must deal with the two implicit coercions appearing in its statement: formally, it has to be stated as follows:

$$i(G(\widehat{x})) = R(j(\widehat{x})) \ ,$$

where i takes a guarded string language and returns a finite word language on the alphabet $\Sigma \uplus \Theta \uplus \Theta$, and j takes a KAT expression and returns a regular expression over this extended alphabet, by pushing all negations to the leaves.

Apart from the properties of these coercion functions, the proof of (ii) mainly consists in rather technical arguments about regular and guarded string languages concatenation. All in all, once KA completeness has been proved, KAT completeness requires us 278 lines of specifications, and 360 lines of proofs.

3 Decision procedure

To check whether two expressions denote the same language of guarded strings, we use an algorithm based on a notion of *partial derivatives* for KAT expressions. Derivatives were introduced by Brzozowski [10] for regular expressions; they make it possible to define a deterministic automaton where the states of the automaton are the regular expressions themselves.

Derivatives can be extended to KAT expressions in a very natural way [22]: we first define a Boolean function ϵ_α, that indicates whether an expression accepts the single atom α; this function is then used to define the derivation function $\delta_{\alpha,p}$, that intuitively returns what remains of the given expression after reading the atom α and the letter p. These two functions make it possible to give a

$$\begin{aligned}
\delta'_{\alpha,p}(x+y) &= \delta'_{\alpha,p}(x) \cup \delta'_{\alpha,p}(y) \\
\delta'_{\alpha,p}(xy) &= \begin{cases} \delta'_{\alpha,p}(x)y \cup \delta'_{\alpha,p}(y) & \text{if } \epsilon_\alpha(x) \\ \delta'_{\alpha,p}(x)y & \text{otherwise} \end{cases} \\
\delta'_{\alpha,p}(x^\star) &= \delta'_{\alpha,p}(x)x^\star
\end{aligned}
\qquad
\begin{aligned}
\delta'_{\alpha,p}(q) &= \begin{cases} \{1\} & \text{if } p = q \\ \emptyset & \text{otherwise} \end{cases} \\
\delta'_{\alpha,p}([a]) &= \emptyset
\end{aligned}$$

Fig. 1. Partial derivatives for KAT expressions

coalgebraic characterisation of the function G, which underpins the correctness of the algorithm we sketch below:

$$G(x)(\alpha) = \epsilon_\alpha(x) \qquad\qquad G(x)(\alpha\, p\, u) = G(\delta_{\alpha,p}(x))(u) \ .$$

Like with standard regular expressions, the set of derivatives of a given KAT expression (i.e., the set of expressions that can be obtained by repeatedly deriving w.r.t. arbitrary atoms and letters) can be infinite. To recover finiteness, we switch to *partial* derivatives [4]. Their generalisation to KAT should be folklore; we define them in Fig. 1. We use the notation Xy to denote the set $\{xy \mid x \in X\}$ when X is a set of expressions and y is an expression. The partial derivation function $\delta'_{\alpha,p}$ returns a (finite) set of expressions rather than a single one; this corresponds to the fact that we build a non-deterministic automaton. Still abusing notations, by letting a set of expressions denote the sum of its elements, we prove that $\mathrm{KAT} \vdash \delta_{\alpha,p}(x) = \delta'_{\alpha,p}(x)$.

Now call *bisimulation* any relation R between sets of expressions such that whenever $X\ R\ Y$, we have

- $\epsilon(X) = \epsilon(Y)$ and
- $\forall \alpha \in At, \forall p \in \Sigma,\ \delta'_{\alpha,p}(X)\ R\ \delta'_{\alpha,p}(Y)$.

We show that if there is a bisimulation R such that $X\ R\ Y$, then $G(X) = G(Y)$ (the converse also holds). This gives us an algorithm to decide language equivalence of two KAT expressions x, y: it suffices to try to construct a bisimulation that relates the singletons $\{x\}$ and $\{y\}$. This algorithm terminates because the set of partial derivatives reachable from a pair of expressions is finite (we do not need to formalise this fact since we just need the correctness of this algorithm).

There is a lot of room for optimisation in our implementation—for instance, we use unordered lists to represent binary relations. An important point in our design is that such optimisations can be introduced and proved correct independently from the completeness proof for KAT, which gives us much more flexibility than in our previous work on Kleene algebra [9].

3.1 Building a reflexive tactic

Using standard methodology [1, 8, 14], we finally pack the previous ingredients into a Coq reflexive tactic called `kat`, allowing us to close automatically any goal which belongs to the equational theory of KAT.

The tactic works on any model of KAT: those already declared in the library (relations, languages, matrices, traces), but also the ones declared by the user. The reification code is written in OCaml; it is quite complicated for at least two reasons: KAT is a two-sorted structure, and we actually deal with "typed" KAT, as explained in §1.2, which requires us to work with a dependently typed syntax.

For the sake of simplicity, the Coq algorithm we implemented for KAT does not produce a counter-example in case of failure. To be able to give such a counter-example to the user, we actually run an OCaml copy of the algorithm first (extracted from Coq, and modified by hand to produce counter-examples). This has two advantages: the tactic is faster in case of failure, and the counter-example—a guarded string—can be pretty-printed in a nicer way.

4 Eliminating hypotheses

The above `kat` tactic works for the equational theory of KAT, i.e., the (in)equations that hold in any model of KAT, under any interpretation. In particular, this tactic does not make use of any hypothesis which is specific to the model or to the interpretation. Some hypotheses can however be exploited [11,15]: those having one of the following shapes.

(i) $x = 0$;
(ii) $[a]x = x[b]$, $[a]x \leq x[b]$, or $x[b] \leq [a]x$;
(iii) $x \leq [a]x$ or $x \leq x[a]$
(iv) $a = b$ or $a \leq b$;
(v) $[a]p = [a]$ or $p[a] = [a]$, for atomic p ($p \in \Sigma$);

Equations of the first kind (i) are called "Hoare" equations, for reasons to become apparent in §5.2. They can be eliminated using the following implication:

$$\begin{cases} x + uzu = y + uzu \\ z = 0 \end{cases} \quad \text{entails} \quad x = y \ . \tag{†}$$

This implication is valid for any term u, and the method is complete [15] when u is taken to be the universal KAT expression, $\Sigma^\star$. Intuitively, for this choice of u, uzu recognizes all guarded strings that contain a guarded string of z as a substring. Therefore, when checking that $x + uzu = y + uzu$ are language equivalent rather than $x = y$, we rule out all counter-examples to $x = y$ that contain a substring belonging to z: such counter-examples are irrelevant since z is known to be empty.

Equations of the shape (iii) and (iv) are actually special cases of those of the shape (ii), which are in turn equivalent to Hoare equations. For instance, we have $[a]x \leq x[b]$ iff $[a]x[\neg b] = 0$. Moreover, two hypotheses of shape (i) can be merged into a single one using the fact that $x = 0 \wedge y = 0$ iff $x + y = 0$. Therefore, we can aggregate all hypotheses of shape (i-iv) into a single one (of shape (i)), and use the above technique just once.

Hypotheses of shape (v) are handled differently, using the following equivalence:

$$[a]p = [a] \quad \text{iff} \quad p = [\neg a]p + [a] \ , \qquad (\ddagger)$$

This equivalence allows us to substitute $[\neg a]p + [a]$ for p in the considered goal—whence the need for p to be atomic. Again, the method is complete [15], i.e.,

$$\text{KAT} \vdash ([a]p = [a] \Rightarrow x = y) \quad \text{iff} \quad \text{KAT} \vdash x\theta = y\theta \qquad (\theta = \{p \mapsto [\neg a]p + [a]\})$$

4.1 Automating elimination of hypotheses in Coq

The previous techniques to eliminate some hypotheses in KAT can be easily automated in Coq. We first prove once and for all the appropriate equivalences and implications (the tactic `kat` is useful for that). We then define some tactics in Ltac that collect hypotheses of shape (i-iv), put them into shape (i), and aggregate them into a single one which is finally used to update the goal according to (†). Separately, we define a tactic that rewrites in the goal using all hypotheses of shape (v), through (‡). Finally, we obtain a tactic called `hkat`, that just preprocesses the conclusion of the goal using all hypotheses of shape (i-v) and then calls the `kat` tactic. Note that the completeness of this method [15] is a meta-theorem; we do not need to formalise it.

5 Case studies

We now present some examples of Coq formalisations where one can take advantage of our library.

5.1 Bigstep semantics of ‘while’ programs

The bigstep semantics of ‘while’ programs is teached in almost any course on semantics and programming languages. Such programs can be embedded into KAT in a straightforward way [21], thus providing us with proper tools to reason about them. Let us formalise such a language in Coq.

Assume a type `state` of states, a type `loc` of memory locations, and an `update` function allowing to update the value of a memory location. Call *arithmetic expression* any function from states to natural numbers, and *Boolean expression* any function from states to Booleans (we use a partially shallow embedding). The ‘while’ programming language is defined by the inductive type below:

```
Variable loc, state: Set.
Variable update: loc → nat → state → state.

Definition expr := state → nat.
Definition test := state → bool.
```

```
Inductive prog :=
| skp
| aff (l: loc) (e: expr)
| seq (p q: prog)
| ite (b: test) (p q: prog)
| whl (b: test) (p: prog).
```

The bigstep semantics of such programs is given as a “state transformer”, i.e., a binary relation between states. Following standard textbooks, one can define this semantics in Coq using an inductive predicate:

```
Inductive bstep: prog → rel state state :=
| s_skp: ∀ s,  bstep skp s s
| s_aff: ∀ l e s,  bstep (aff l e) s (update l (e s) s)
| s_seq: ∀ p q s s' s'',  bstep p s s' → bstep q s' s'' → bstep (seq p q) s s''
| s_ite_ff: ∀ b p q s s',  ¬ b s → bstep q s s' → bstep (ite b p q) s s'
| s_ite_tt: ∀ b p q s s',  b s → bstep p s s' → bstep (ite b p q) s s'
| s_whl_ff: ∀ b p s,  ¬ b s → bstep (whl b p) s s
| s_whl_tt: ∀ b p s s',  b s → bstep (seq p (whl b p)) s s' → bstep (whl b p) s s'.
```

Alternatively, one can define this semantic through the relational model of KAT, by induction over the program structure:

```
Fixpoint bstep (p: prog): rel state state :=
  match p with
    | skp ⇒ 1
    | seq p q ⇒ bstep p·bstep q
    | aff l e ⇒ upd l e
    | ite b p q ⇒ [b]·bstep p + [¬b]·bstep q
    | whl b p ⇒ ([b]·bstep p)⋆·[¬b]
  end.
```

(Notations come for free since binary relations are already declared as a model of KAT in our library.) The 'skip' instruction is interpreted as the identity relation; sequential composition is interpreted by relational composition. Assignments are interpreted using an auxiliary function, defined as follows:

```
Definition upd l e: rel state state := fun s s' ⇒ s' = update l (e s) s.
```

For the 'if-then-else' statement, the Boolean expression b is a predicate on states, i.e., a test in our relational model of KAT; this test is used to guard both branches of the possible execution paths. Accordingly for the 'while' loop, we iterate the body of the loop guarded by the test, using Kleene star. We make sure one cannot exit the loop before the condition gets false by post-guarding the iteration with the negation of this test.

This alternative definition is easily proved equivalent to the previous one. Its relative conciseness makes it easier to read; more importantly, this definition allows us to exploit all theorems and tactics about KAT, for free. For instance, suppose that one wants to prove some program equivalences. First define program equivalence, through the bigstep semantics:

```
Notation "p ∼ q" := (bstep p == bstep q).
```

(The "==" symbol denotes equality in the considered KAT model; in this case, relational equality.) The following lemmas about unfolding loops and dead code elimination, can be proved automatically.

```
Lemma two_loops b p:  whl b (whl b p) ∼ whl b p.
Proof. simpl. kat. Qed.
(* ([b]·(([b]·bstep p)⋆·[¬b]))⋆·[¬b] == ([b]·bstep p)⋆·[¬b] *)

Lemma fold_loop b p:  whl b (p ; ite b p skp) ∼ whl b p.
Proof. simpl. kat. Qed.
(* ([b]·(bstep p·([b]·bstep p + [¬b]·1)))⋆·[¬b] == ([b]·bstep p)⋆·[¬b] *)
```

```
Lemma dead_code a b p q r:   whl (a ∨ b) p ; ite b q r   ∼   whl (a ∨ b) p ; r.
Proof. simpl. kat. Qed.
(* ([a ∨ b]·bstep p)⋆·[¬(a ∨ b)]·([b]·bstep q + [¬b]·bstep r)
                                         == ([a ∨ b]·bstep p)⋆·[¬(a ∨ b)]·bstep r *)
```

(The semicolon in program expressions is a notation for sequential composition; the comments below each proof show the intermediate goal where the `bstep` fixpoint has been simplified, thus revealing the underlying KAT equality.)

Of course, the `kat` tactic cannot prove arbitrary program equivalences: the theory of KAT only deals with the control-flow graph of the programs and with the Boolean expressions, not with the concrete meaning of assignments or arithmetic expressions. We can however mix automatic steps with manual ones. Consider for instance the following example, where we prove that an assignment can be delayed. Our tactics cannot solve it automatically since some reasoning about assignments is required; however, by asserting manually a simple fact (in this case, an equation of shape (ii)), the goal becomes provable by the `hkat` tactic.

```
Definition subst l e (b: test): test := fun s ⇒ b (update l (e s) s).
Lemma aff_ite l e b p q: (l←e; ite b p q)  ∼  (ite (subst l e b) (l←e; p) (l←e; q)).
Proof.
  simpl. (* upd l e·([b]·bstep p + [¬b]·bstep q) ==
         [subst l e b]·(upd l e·bstep p)·[¬subst l e b]·(upd l e·bstep q) *)
  assert (upd l e·[b]  == [subst l e b]·upd l e) by (cbv; firstorder; subst; eauto).
  hkat.
Qed.
```

5.2 Hoare logic for partial correctness

Hoare logic for partial correctness [16] is subsumed by KAT [21]. The key ingredient in Hoare logic is the notion of a "Hoare triple" $\{A\}\,p\,\{B\}$, where p is a program, and A, B are two formulas about the memory manipulated by the program, respectively called pre- and post-conditions. A Hoare triple $\{A\}\,p\,\{B\}$ is *valid* if whenever the program p starts in some state s satisfying A and terminates in a state s', then s' satisfies B. Such a statement can be translated into KAT as a simple equation:

$$[A]p[\neg B] = 0$$

Indeed, $[A]p[\neg B] = 0$ precisely means that there is no execution path along p that starts in A and ends in $\neg B$. Such equations are Hoare equations (they have the shape (i) from §4), so that they can be eliminated automatically. As a consequence, inference rules of Hoare logic can be proved automatically using the `hkat` tactic. For instance, for the 'while' rule, we get the following script:

```
Lemma rule_whl A b p: {A ∧ b} p {A} → {A} whl b p {A ∧ ¬b}.
Proof. simpl. hkat. Qed.
(* [A ∧ b]·bstep p·[¬A] == 0 → [A]·(([b]·bstep p)⋆·[¬b])·[¬(A ∧ ¬b)] == 0 *)
```

5.3 Compiler optimisations

Kozen and Patron [23] use KAT to verify a rather large range of standard compiler optimisations, by equational reasoning. Citing their abstract, they cover "*dead code elimination, common subexpression elimination, copy propagation, loop hoisting, induction variable elimination, instruction scheduling, algebraic simplification, loop unrolling, elimination of redundant instructions, array bounds check elimination, and introduction of sentinels*". They cannot use automation, so that the size of their proofs ranges from a few lines to half a page of KAT computations.

We formalised all those equational proofs using our library. Most of them can actually be solved instantaneously, by a simple call to the `hkat` tactic. For the few remaining ones, we gave three to four lines proofs, consisting of first rewriting using hypotheses that cannot be eliminated, and then a call to `hkat`.

The reason why `hkat` performs so well is that most assumptions allowing to optimise the code in these examples are of the shape (i-v). For instance, to state that an instruction p has no effect when $[a]$ is satisfied, we use an assumption $[a]p = [a]$. Similarly, to state that the execution of a program x systematically enforces $[a]$, we use an assumption $x = x[a]$. The assumptions that cannot be eliminated are typically those of the shape $pq = qp$: "the instructions p and q commute"; such assumptions have to be used manually.

5.4 Flowchart schemes

The last example we discuss here is due to Paterson, it consists in proving the equivalence of two flowchart schemes (i.e., goto programs—see Manna's book [26] for a complete description of this model). The two schemes are given in Appendix A; Manna proves their equivalence using several successive graph transormations. His proof is really high-level and informal; it is one page long, plus three additional pages to draw intermediate flowcharts schemes. Angus and Kozen [3] give a rather detailed equational proof in KAT, which is about six pages long. Using the `hkat` tactic together with some ad-hoc rewriting tools, we managed to formalise Angus and Kozen's proof in three rather sparse screens.

Like in Angus and Kozen's proof, we progressively modify the KAT expression corresponding to the first schema, to make it evolve towards the expression corresponding to the second schema. Our mechanised proof thus roughly consists in a sequence of transitivity steps closed by `hkat`, allowing us to perform some rewriting steps manually and to move to the next step. This is illustrated schematically by the code presented in Fig. 2.

Most of our transitivity steps (the y_i's) already appear in Angus and Kozen's proof; we can actually skip a lot of their steps, thanks to `hkat`. Some of these simplifications can be spectacular: for instance, they need one page to justify the passage between their expressions (24) and (27), while a simple call to `hkat` does the job; similarly for the page they need between their steps (38) and (43).

```
Lemma Paterson: x_1 == z.
Proof.
  transitivity y_1. hkat.        (* x_1 == y_1 *)
  a few rewriting steps transforming y_1 into x_2.
  transitivity y_2. hkat.        (* x_2 == y_2 *)
  a few rewriting steps transforming y_2 into x_3.
  (* ... *)
  transitivity y_19. hkat.       (* x_19 == y_19 *)
  a few rewriting steps transforming y_19 into x_20.
  hkat.                          (* x_20 == z *)
Qed.
```

Fig. 2. Squeleton for the proof of equivalence of Paterson's flowchart schems

6 Related works

Several formalisations of algorithms and results related to regular expressions and languages have been proposed since we released our Coq reflexive decision procedure for Kleene algebra [9]: partial derivatives for regular expressions [2], regular expression equivalence [6, 12, 25, 27], regular expression matching [17]. None of these works contains a formalised proof of completeness for Kleene algebra, so that they cannot be used to obtain a general tactic for KA (note however that Krauss and Nipkow [25] obtain an Isabelle/HOL tactic for binary relations using a nice trick to sidestep the completeness proof—but they cannot deal with other models of KA).

On the more algebraic side, Struth et al. [5, 13] showed how to formalise and use relation algebra and Kleene algebra in Isabelle/HOL; they exploit the automation tools provided by this assistant, but they do not try to define decision procedures specific to Kleene algebra, and they do not prove completeness.

To the best our knowledge, the only formalisation of KAT prior to the present work is due to Pereira and Moreira [28], in Coq. They state all axioms of KAT, derive some simple consequences of these axioms (e.g., Boolean disjunction distribute over conjunction, Kleene star is monotone), and use them to manually prove the inference rules of Hoare logic, as we did automatically in §5.2. They do not provide models, automation tools, or completeness proof.

7 Conclusion

We presented a rather exhaustive Coq formalisation of Kleene algebra with tests: axiomatisation, models, completeness proof, decision procedure, elimination of hypotheses. We then showed several use-cases for the corresponding library: proofs about while programs and Hoare logic, certification of standard compiler optimisations, and equivalence of flowchart schemes.

Most of the theoretical material is due to Kozen et al. [3, 15, 18–24], so that our contribution mostly lies in the Coq mechanisation of these ideas. The completeness proof was particularly challenging to formalise, and lots of aspects of this work could not be explained in this extended abstract: how to encode the algebraic hierarchy, how to work efficiently with finite sets and finite sums, how to exploit symmetry arguments, reflexive normalisation tactics, tactics about lattices, finite ordinals and encodings of set-theoretic constructs in ordinals…

The Coq library is available online [30]; it is documented and axiom-free; its overall structure is given in Appendix B. This library actually has a larger scope than what we presented here: our long-term goal is to formalise and automate other fragments of relation algebra (residuated structures, Kleene algebra with converse, allegories…), so that the library is designed to allow for such extensions. For instance normalisation tactics and an ad-hoc semi-decision procedures are already defined for algebraic structures beyond Kleene algebra and KAT.

According to `coqwc`, the library consists of 4377 lines of specifications and 3020 lines of proofs, that distribute as follows. Overall, this is slightly less than our previous library for KA [9] (5105+4315 lines), and we do much more: not only we handle KAT, but we also lay the ground for the mechanisation of other fragments of relation algebra, as explained above.

	specifications	proofs	comments
ordinals, comparisons, finite sets…	674	323	225
algebraic hierarchy	490	374	216
models (languages, relations, expressions…)	1279	461	404
linear algebra, matrices	534	418	163
completeness, decisions procedure, tactics	1400	1444	740

The resulting theorems and tactics allowed us to shorten significantly a number of paper proofs—those about Hoare logic, compiler optimisations, and flowchart schemes. Getting a way to guarantee that such proofs are correct is important: although mathematically simple, they tend to be hard to proofread (we invite the skeptical reader to check Angus and Kozen's paper proof of Paterson example [3]). Moreover, automation greatly helps when searching for such proofs: being able to get either a proof or a counter-example for any proposed equation is a big plus: it makes it much easier to progress in the overall proof.

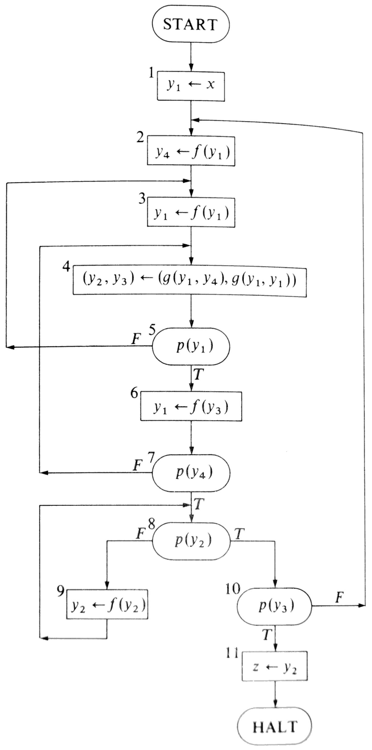
References

1. S. F. Allen, R. L. Constable, D. J. Howe, and W. E. Aitken. The semantics of reflected proof. In *Proc. LICS*, pages 95–105. IEEE Computer Society, 1990.
2. J. B. Almeida, N. Moreira, D. Pereira, and S. M. de Sousa. Partial derivative automata formalized in Coq. In *Proc. CIAA*, volume 6482 of *LNCS*, pages 59–68. Springer, 2010.
3. A. Angus and D. Kozen. Kleene algebra with tests and program schematology. Technical Report TR2001-1844, CS Dpt, Cornell University, July 2001.
4. V. M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *TCS*, 155(2):291–319, 1996.
5. A. Armstrong and G. Struth. Automated reasoning in higher-order regular algebra. In *Proc. RAMiCS*, volume 7560 of *LNCS*, pages 66–81. Springer, 2012.

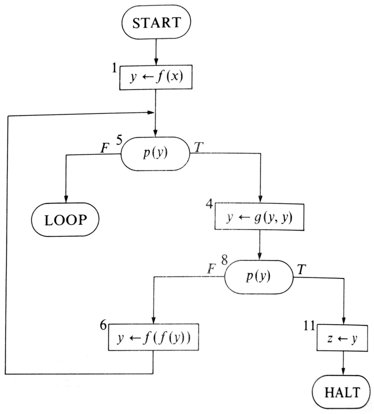
6. A. Asperti. A compact proof of decidability for regular expression equivalence. In *Proc. ITP*, volume 7406 of *LNCS*, pages 283–298. Springer, 2012.
7. Y. Bertot, G. Gonthier, S. O. Biha, and I. Pasca. Canonical big operators. In *TPHOLs*, volume 5170 of *LNCS*, pages 86–101. Springer, 2008.
8. R. Boyer and J. Moore. Metafunctions: proving them correct and using them efficiently as new proof procedures. In *The Correctness Problem in Computer Science*. NY: Academic Press, 1981.
9. T. Braibant and D. Pous. An efficient Coq tactic for deciding Kleene algebras. In *Proc. 1st ITP*, volume 6172 of *LNCS*, pages 163–178. Springer, 2010.
10. J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.
11. E. Cohen. Hypotheses in Kleene algebra. Technical report, Bellcore, Morristown, N.J., 1994.
12. T. Coquand and V. Siles. A decision procedure for regular expression equivalence in type theory. In *Proc. CPP*, volume 7086 of *LNCS*. Springer, 2011.
13. S. Foster and G. Struth. Automated analysis of regular algebra. In *Proc. IJCAR*, volume 7364 of *LNCS*, pages 271–285. Springer, 2012.
14. B. Grégoire and A. Mahboubi. Proving equalities in a commutative ring done right in Coq. In *Proc. TPHOL*, volume 3603 of *LNCS*, pages 98–113. Springer, 2005.
15. C. Hardin and D. Kozen. On the elimination of hypotheses in Kleene algebra with tests. Technical Report TR2002-1879, CS Dpt, Cornell University, October 2002.
16. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
17. V. Komendantsky. Reflexive toolbox for regular expression matching: verification of functional programs in Coq+ssreflect. In *Proc. PLPV*, pages 61–70. ACM, 2012.
18. D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Inf. and Comp.*, 110(2):366–390, 1994.
19. D. Kozen. Kleene algebra with tests. *Transactions on Programming Languages and Systems*, 19(3):427–443, May 1997.
20. D. Kozen. Typed Kleene algebra, 1998. TR98-1669, CS Dpt. Cornell University.
21. D. Kozen. On Hoare logic and Kleene algebra with tests. *ACM Trans. Comput. Log.*, 1(1):60–76, 2000.
22. D. Kozen. On the coalgebraic theory of Kleene algebra with tests. Technical Report http://hdl.handle.net/1813/10173, CIS, Cornell University, March 2008.
23. D. Kozen and M.-C. Patron. Certification of compiler optimizations using Kleene algebra with tests. In *Proc. CL2000*, volume 1861 of *LNAI*, pages 568–582. Springer, 2000.
24. D. Kozen and F. Smith. Kleene algebra with tests: Completeness and decidability. In *Proc. CSL*, volume 1258 of *LNCS*, pages 244–259. Springer, September 1996.
25. A. Krauss and T. Nipkow. Proof pearl: Regular expression equivalence and relation algebra. *JAR*, 49(1):95–106, 2012.
26. Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.
27. N. Moreira, D. Pereira, and S. M. de Sousa. Deciding regular expressions (in-)equivalence in Coq. In *Proc. RAMiCS*, volume 7560 of *LNCS*, pages 98–113. Springer, 2012.
28. D. Pereira and N. Moreira. KAT and PHL in Coq. *Comput. Sci. Inf. Syst.*, 5(2):137–160, 2008.
29. D. Pous. Untyping typed algebraic structures and colouring proof nets of cyclic linear logic. In *Proc. CSL*, volume 6247 of *LNCS*, pages 484–498. Springer, 2010.
30. D. Pous. RelationAlgebra: Coq library containing all material presented in this paper. http://perso.ens-lyon.fr/damien.pous/ra, December 2012.
31. K. Thompson. Regular expression search algorithm. *C. ACM*, 11:419–422, 1968.

A Paterson's flowchart schemes

Here are the two flowchart schemes we proved equivalent (§5.4), they appear in [26, pages 254 and 258].



Schema S_{6A}



Schema S_{6E}

Following Angus and Kozen's notations [3], these two schemes can be converted into the following KAT expressions:

$$\begin{aligned}S_{6A} = {} & x_1 p_{41} p_{11} q_{214} q_{311} \left([\neg a_1] p_{11} q_{214} q_{311}\right)^\star [a_1] p_{13} \\ & \Big(([\neg a_4] + [a_4]([\neg a_2] p_{22})^\star [a_2 \wedge \neg a_3] p_{41} p_{11})\, q_{214} q_{311} \left([\neg a_1] p_{11} q_{214} q_{311}\right)^\star [a_1] p_{13}\Big)^\star \\ & [a_4] \left([\neg a_2] p_{22}\right)^\star [a_2 \wedge a_3] z_2\end{aligned}$$

$$S_{6E} = s_1 [a_1] q_1 \left([\neg a_1] r_1 [a_1] q_1\right)^\star [a_1] z_1 \ \ ,$$

where the tests and actions are interpreted as follows:

$$x_i \triangleq y_i \leftarrow x \qquad z_i \triangleq z \leftarrow y_i \qquad a_i \triangleq P(y_i)$$
$$p_{ij} \triangleq y_i \leftarrow f(y_j) \qquad q_{ijk} \triangleq y_i \leftarrow g(y_j, y_k)$$

(Note that we actually renamed the local variable y from schema S_{6E} into y_1, for the sake of uniformity.)

B Overall structure of the library

Here is a succinct description of each module from the library:

Utilities

- `common`: basic tactics and definitions used throughout the library
- `comparisons`: types with decidable equality and ternary comparison function
- `positives`: simple facts about binary positive numbers
- `ordinal`: finite ordinals, finite sets of finite ordinals
- `pair`: encoding pairs of ordinals as ordinals
- `powerfix`: simple pseudo-fixpoint iterator
- `lset`: sup-semilattice of finite sets represented as lists

Algebraic hierarchy

- `level`: bitmasks allowing us to refer to an arbitrary point in the hierarchy
- `lattice`: “flat” structures, from preorders to Boolean lattices
- `monoid`: typed structures, from po-monoids to residuated Kleene lattices
- `kat`: Kleene algebra with tests
- `kleene`: Basic facts about Kleene algebra
- `normalisation`: normalisation and semi-decision tactics for relation algebra

Models

- `prop`: distributive lattice of propositions
- `boolean`: Boolean trivial lattice, extended to a monoid.
- `rel`: heterogeneous binary relations
- `lang`: word languages
- `traces`: trace languages
- `atoms`: atoms of the free Boolean lattice over a finite set
- `glang`: guarded string languages
- `lsyntax`: free lattice (Boolean expressions)
- `syntax`: free relation algebra
- `regex`: regular expressions
- `gregex`: KAT expressions (typed—for KAT completeness)
- `ugregex`: untyped KAT expressions (untyped—for KAT decision procedure)

Untyping theorems

- `untyping`: untyping theorem for structures below KA with converse
- `kat_untyping`: untyping theorem for guarded string languages

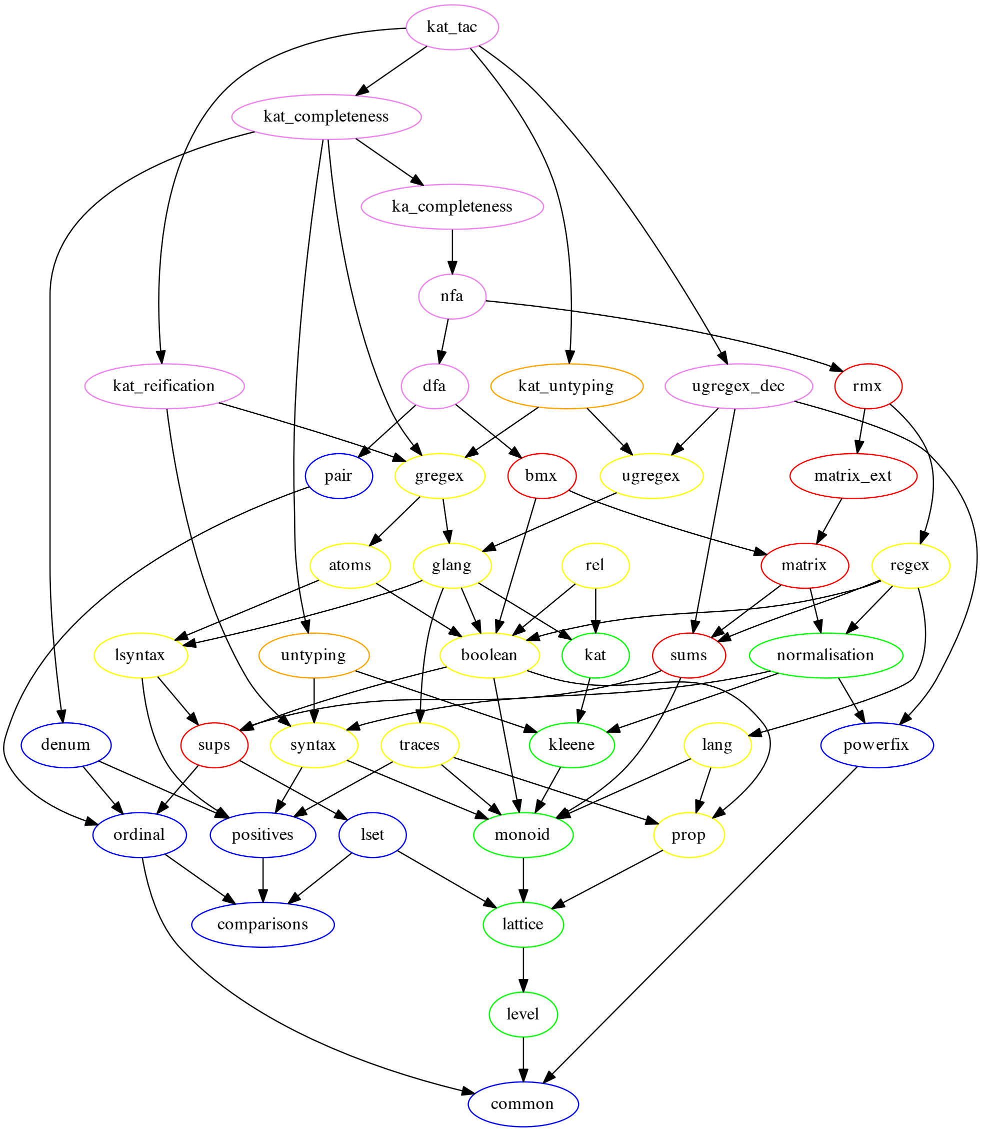
Linear algebra

- `sups`: finite suprema/infima (a la bigop, from ssreflect)
- `sums`: finite sums
- `matrix`: matrices over all structures supporting this construction
- `matrix_ext`: additional operations and properties about matrices
- `rmx`: matrices of regular expressions
- `bmx`: matrices of Booleans

Automata, completeness

- `dfa`: deterministic finite state automata, decidability of language inclusion
- `nfa`: matricial non-deterministic finite state automata
- `ugregex_dec`: decision of language equivalence for KAT expressions
- `ka_completeness`: (untyped) completeness of Kleene algebra
- `kat_completeness`: (typed) completeness of Kleene algebra with tests
- `kat_reification`: tools and definitions for KAT reification
- `kat_tac`: decision tactics for KA and KAT, elimination of hypotheses

Here are the dependencies between these modules:


kat_tac
kat_completeness
ka_completeness
nfa
kat_reification
dfa
kat_untyping
ugregex_dec
rmx
pair
gregex
bmx
ugregex
matrix_ext
atoms
glang
rel
matrix
regex
lsyntax
untyping
boolean
kat
sums
normalisation
denum
sups
syntax
traces
kleene
lang
powerfix
ordinal
positives
lset
monoid
prop
comparisons
lattice
level
common