Certified Intersection of Commutative Regular Expressions as Solutions of Systems of Linear Diophantine Equations

Ricardo Almeida
School of Computing Science, University of Glasgow, Glasgow, Scotland

Blair Archibald
School of Computing Science, University of Glasgow, Glasgow, Scotland

Basile Pesin
Fédération ENAC ISAE-SUPAERO ONERA, Université de Toulouse, France

Michele Sevegnani
School of Computing Science, University of Glasgow, Glasgow, Scotland

Abstract

Commutative regular expressions describe sets of *unordered* words, and are used, for example, when building type systems for process calculi. In these applications, an important operation is finding the intersection of two expressions, but no algorithm currently exists. We remedy this by proposing an algorithm for computing intersections of commutative regular expressions, which we implement and prove correct in the Rocq prover. The algorithm encodes the intersection of two expressions as systems of linear Diophantine equations, and extracts from their solution an intersection expression. To solve these systems we implement and verify the algorithm proposed by Contejean and Devie. We detail the implementation of the intersection algorithm, highlight essential aspects of the proofs (including the complex proof of termination of the equation system solver), and evaluate the OCaml-extracted solver on random and real-world commutative regular expressions.

2012 ACM Subject Classification Theory of computation → Regular languages; Software and its engineering → Formal software verification

Keywords and phrases commutative regular expressions, linear Diophantine equations, interactive theorem provers, Rocq

Digital Object Identifier 10.4230/LIPIcs.ITP.2026.8

Supplementary Material *Software (Rocq Artefact)*: `https://doi.org/10.5281/zenodo.20265929`

Funding This work is partially supported by the British Council through the Alliance Hubert Curien Programme under grant 49780UL and an Amazon Research Award on Automated Reasoning.

Acknowledgements We thank A. Logan for pointing us in the direction of the Contejean-Devie algorithm for solving systems of linear Diophantine equations, and Rocq-discourse users (A. Trieu, D. Larchey-Wendling, Y. Zakowski and J-M. Mariot) for their help with the proof of Lemma 8.

1 Introduction

Commutative regular expressions denote sets of words where the letters are said to commute: for instance, the expression $(ab)^*$ denotes the language of all words that contain an equal number of a and b, and no other letter. They appear in different fields of computer science, but often under distinct names: commutative regular grammars [19], multiset regular expressions [12], unordered regular expressions (with applications to XML types [5]), regular bag expressions (used in RDF schemas [32]), etc. More recently, commutative regular expressions have also appeared as types for process calculi and concurrent programming.

Mailbox types [13, 15] describe unordered interactions between two processes. For instance, $?(a^*)(b^*)c$ is the type of a process that can receive any number of a and b messages, and must receive exactly one c message, regardless of the order. Another calculus, Milner's bigraphs [23], may be equipped with *sorts* to express how bigraphs may be composed and permit only well-formed instances. Mailbox types have been adapted to specify sorting rules for a bigraphical language, and the well-sortedness of several constructs is based on the existence of an intersection of two sort expressions [4]. However, the authors do not specify how to compute this intersection.

More generally, to our knowledge, there is no published work on computing the intersection of two commutative regular expressions (i.e., an expression that recognises exactly the words recognised by both input expressions). In Conway's words, "Commutative regular algebra is notable for the number of results whose proofs one would expect to be trivial, but which turn out to be very subtle" [11]. The commutativity axiom renders many results and algorithms for traditional (i.e., non-commutative) regular expressions inapplicable to the commutative case. Many of these algorithms rely on first converting the expressions to automata and then performing operations on automata directly. This is the case of intersection for standard regular expressions, which can be computed via the product of their automata representations (or, following De Morgan's laws, via the complement of the union of the two automata complemented) [18]. Since automata naturally enforce an order in which letters are read, these results do not transfer to the commutative case. While in 1971 Conway did prove that commutative regular expressions are closed under intersection [11], this proof is not only non-constructive but also somewhat informal in several steps. Therefore, an algorithm for computing the intersection of commutative expressions was still missing.

Since words in the language of a commutative expression are defined only by the number of occurrences of each letter, our algorithm first calculates equality constraints between letter occurrences in the two input expressions e_1 and e_2. Resolving these constraints amounts to finding the set of minimal solutions of a non-homogeneous system of linear Diophantine equations. Based on Fortenbacher's algorithm for solving a single linear Diophantine equation [6], Contejean and Devie proposed one of the most general algorithms in the literature, solving not just one equation but a system with arbitrarily many equations that may, additionally, be non-homogeneous [10]. Since it remains one of the most widely used solvers (e.g., it is one of the two implementations in the Maude system's built-in linear Diophantine equation solver [8, 7]), we chose this particular algorithm to formalise in our pipeline. While Meßner et al. formalised a solver for single homogeneous linear Diophantine equations in Isabelle/HOL [22], to the best of our knowledge, our work presents the first formalisation for the general case of systems of (non-)homogeneous linear equations. We extract the certified solver to OCaml code, which may be used either independently or in integration with other tools. Systems of linear Diophantine equations have their own applications in diverse fields, such as Petri nets [24], constraint logic programming [1], decomposing chemical reactions [25], and unification modulo associativity [21], commutativity [33, 17], and distributivity [9].

Our paper and contributions follow the structure highlighted in Figure 1 and described below. Our entire formalisation and examples are provided as an artefact [3]. It also includes the first benchmark of intersections of commutative regular expressions.

- Section 2 presents our first contribution: a **mechanisation of commutative regular expressions with denotational semantics** ($[\![e]\!]$ denotes the set of all multisets of letters recognised by e), and establishes the correspondence with the axiomatic semantics of [11].

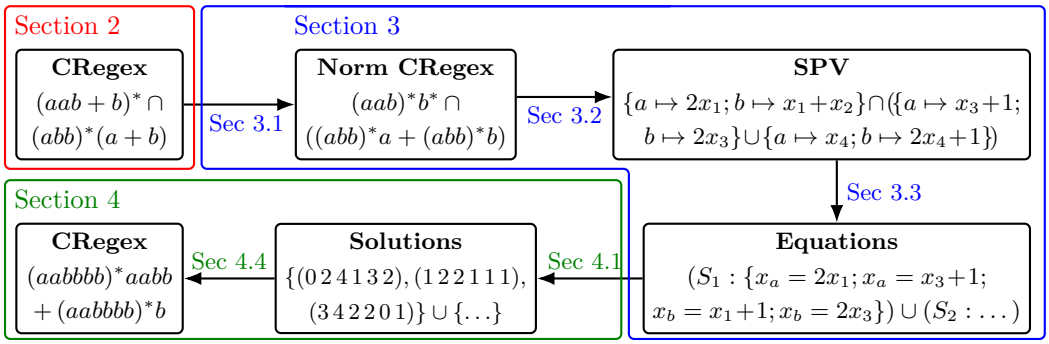

Figure 1 Intersection algorithm pipeline with an illustrative intersection example.

- Section 3 details the first half of the intersection algorithm: a series of transformations encoding the intersection constraints between two expressions as equation systems.
- Section 4 then presents our second contribution : the first **mechanisation of a solver for (non-)homogeneous systems of linear Diophantine equations**, based on the algorithm proposed by Contejean and Devie. It discusses the challenges around its proofs of termination and correctness. Finally, we put it all together in our third and main contribution: the first **intersection algorithm for commutative regular expressions**, implemented as a pure Rocq function `intersection`, and close with the final result: $[\![\texttt{intersection}\ e_1\ e_2]\!] \equiv [\![e_1]\!] \cap [\![e_2]\!]$.
- Section 5 evaluates the performance of our implementation using both uniformly random generated expressions and real-world expressions.
- Section 6 discusses the development as a whole and presents directions for future work.

2 A Formalisation of Commutative Regular Expressions

Let $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ be an *alphabet*. A *word over* Σ is a finite sequence of letters, and in a commutative word all letters are mutually commuting by definition (e.g., $ab = ba$). Since only the number of occurrences of each letter matters (and not the order), a commutative word can be identified by a multiset of letters, or Parikh Image (PI), that maps each letter to its number of occurrences and whose type is given by: `Definition PI (Σ: Type) := Σ → nat`. In the following definitions, we fix the alphabet Σ. A *commutative regular expression* defines a *language*, i.e. a set of commutative words, and can be obtained from 0, 1, and letters in Σ by a finite application of the regular operations $\cdot$ (often omitted), $+$ and *.

We formalise the *commutative regular expressions* over Σ as an inductive type in Listing 1. The language acts as denotational semantics for expressions. Sets are defined by their membership condition, and a language is a set of PIs over the alphabet Σ (which we will simply call the PI of that language). Then, `lan e` computes the language associated with expression `e`. In its presentation, we use subscript n to differentiate multiset notations from set notations (e.g. $\emptyset_n$ is the empty multiset while $\emptyset$ is the empty set). Expression `Zero` does not recognise any word, `One` recognises only the empty word, and `Letter x` recognises only the word with exactly one occurrence of `x`. As expected, `Sum` recognises the union of the languages recognised by its arguments. The `Prod` operator is a bit more difficult to specify: `Prod e f` recognises any word that is the sum of a word recognised by `e` and a word recognised by `f`. Formally: $s_1 \oplus s_2 = \{m \mid \exists m_1 \in s_1, m_2 \in s_2, \forall \sigma, m(\sigma) = m_1(\sigma) + m_2(\sigma)\}$. Finally, `Star e` recognises any word recognised by any k-ary product of `e` by applying $\oplus$ k times.

```
Inductive exp :=
| Zero | One | Letter (x: Σ) | Prod (e f: exp) | Sum (e f: exp) | Star (e: exp).

Definition set (A: Type) := A → ℙ .
Definition lang : Type := set (PI Σ).
Fixpoint lan (e: exp) : lang :=
  match e with
  | Zero ⇒ ∅
  | One ⇒ {∅_n}
  | Letter x ⇒ {{x}_n}
  | Sum e f ⇒ (lan e) ∪ (lan f)
  | Prod e f ⇒ (lan e) ⊕ (lan f)
  | Star e ⇒ λpi ⇒ ∃k, pi ∈ ⨁^k (lan e)
  end.
```

Listing 1 Commutative regular expressions and their language.

2.1 Denotational and Axiomatic Semantics

To check that our denotational semantics faithfully represent the behavior of commutative regular expressions as understood in the litterature, we proved its correspondence with an earlier axiomatic definition.

Two expressions are said to be equivalent if they recognise the same language, which we write `lan e` $\equiv$ `lan f`. Redko [28] and Salomaa [30] proposed an axiomatization of commutative regular expressions, stating that any equivalence between two expressions is a consequence of the classical axioms of regular expressions [11] extended with axioms 1 and 2 below [28] (axiom (3) can be derived from (2) [11]). This became known as *Redko's theorem* [28], whose proof was completed by Pilling [26] and later simplified by Conway [11].

$$ef = fe \quad (1) \qquad e^*f^* = (ef)^*(e^* + f^*) \quad (2) \qquad (e + f)^* = e^*f^* \quad (3)$$

We have proven that `lan` satisfies all classical and commutative axioms (see the lemma for axiom (2) below), some of which can be derived from earlier ones. This result is the converse of Redko's theorem: since the $\equiv$ relation is transitive, the application of any combination of rewriting axioms yields an equivalent expression. It remains to be seen if we could also use these definitions to formalise Redko's theorem. In addition to this theoretical result, these rules are also useful in proving the `intersection` function described in the next sections.

```
Lemma star_prod : ∀e f,
  lan (Prod(Star e)(Star f)) ≡ lan (Prod(Star(Prod e f))(Sum(Star e)(Star f))).
```

Listing 2 Rocq statement of Equation (2).

3 Intersection as Systems of Linear Diophantine Equations

This section describes the first half of the algorithm that computes the intersection of two commutative regular expressions. Given two arbitrary expressions, it computes a disjunction of systems of linear Diophantine equations. These equations describe the relation between the number of occurrences of each letter in the language of the intersection. The algorithm is defined as a sequence of three transformations:

1. normalise each input expression

2. compute a disjunction of Symbolic Parikh Vectors (SPVs) from each normalised expression
3. compute a system of linear Diophantine equations for each possible pair of SPVs

For each of the intermediate constructs used in the algorithm (SPVs, equations, Diophantine equations), we defined an interpretation in terms of the language recognised by the object. The proof of correctness of each transformation then amounts to proving that the languages recognised by the source and target of the transformation are equal. We now describe each transformation in more detail, along with elements of the associated correctness proof.

3.1 Normalisation

The first pass of the algorithm consists of simplifying expressions into a form from which equational constraints may be extracted to define the intersection problem. Conway's "Normal form theorem" [11, Chap 11, Thm 1] states that "any commutative regular expression (...) can be expressed in normal form as a finite sum of terms of the shape $w_1^*...w_n^*w$, where $w_1, ..., w_n, w$ are words (i.e., products) of the letters".

In Rocq, we specify Conway's normal form as the inductive definition `dnf` shown in Listing 3, and we implemented the `normalise` algorithm that transforms an arbitrary expression into an expression respecting the `dnf` shape. It proceeds by applying both classical and commutativity axioms (see Section 2.1) in three passes: first, it distributes `Prod` operations over `Sum`, and turns `Star` operations of `Sum` expressions into `Prod` operations of `Star` expressions; second, it removes nested stars; last, it applies neutral elements, associativity and commutativity rules to put the resulting expression into the exact shape specified by `dnf`. For example, expressions $(aab + b)^*$ and $(abb)^*(a + b)$ from Figure 1 have normal forms $(aab)^*b^*$ and $(abb)^*a + (abb)^*b$, respectively.

```
Inductive word: exp → ℙ :=
| w_letter: word (Letter x)
| w_prod: word e → word (Prod (Letter x) e).
Inductive term: exp → ℙ :=
| t_one: term One
| t_word: word e → term e
| t_prod: word e1 → term e2 → term (Prod (Star e1) e2).
Inductive dnf: exp → ℙ :=
| d_zero: dnf Zero
| d_sum: term e1 → dnf e2 → dnf (Sum e1 e2).
```

Listing 3 Normal form for commutative regular expressions.

The two lemmas below prove the "Normal form theorem" constructively. The first specifies that normalising preserves the language recognised by the expression. Its proof mainly consists of applying the axiomatic rules discussed in Section 2.1, to show that each atomic rewriting of the syntax preserves the recognised language.

▶ **Lemma 1.** *Normalisation preserves the recognised language*

```
∀ e, lan (normalise e) ≡ lan e
```

The second specifies that, after normalisation, the expression is indeed a `dnf`. It is structured in three sub-lemmas, one for each normalisation pass. To simplify the statement of these lemmas, we wrote precise inductive definitions of the shape of the expressions after each intermediate pass (e.g. expressions where `Prod` have been distributed). This decomposition increased the number of inductive definitions but simplified stating and proving the lemmas.

▶ **Lemma 2.** *Normalising an expression produces a* `dnf`

```
∀ e, dnf (normalise e)
```

3.2 Translation into Symbolic Parikh Vectors

As explained in Section 2, the language recognised by an expression can be identified by its PI. In this section, we show how to compute a SPV, an abstraction of this PI well-suited to generate intersection equations.

In the previous section, we showed that any expression can be rewritten in a normal form which corresponds to sums of terms $w_1^*...w_n^*w$. We observe that such a term identifies an abstraction over PIs that maps alphabet letters to linear expressions defining their possible number of occurrences: each letter σ_i is mapped to $k_{i,1}x_1 + ... + k_{i,n}x_n + k_i$, where each $k_{i,j}$ denotes the number of occurrences of σ_i in w_j, and k_i is the number of occurrences in the constant term w. The variables $x_1, \ldots, x_n$ range freely over $\mathbb{N}$ since every ${w_j}^*$ corresponds to 0 or more repetitions of w_j. We call this abstract representation a *Symbolic Parikh Vector* (SPV). For instance, the expression $(ab)^*(aab)^*b$ is equivalent to $\{a \mapsto x_1+2x_2+0; b \mapsto x_1+x_2+1\}$. Our intersection algorithm decomposes the sum obtained after normalisation into a list of terms, and returns an SPV for each term. The union of the languages recognised by these SPVs equals that recognised by the source expression.

We present our mechanisation of SPVs in Listing 4 below. An SPV is a partial map from letters (type `Map.t`) to "rows", which encode the linear expressions defined above. To ensure that all rows have the same length (i.e., that the linear expressions all have the same number of variables), we use dependently typed vectors: a `row n` is a pair containing i) a vector `vec ℕ n` of exactly `n` natural numbers (`n` variables) and ii) a natural number (the coefficient of the constant term). The semantics of an SPV are given in terms of sets of PIs, similarly to expressions. The definition `eval_row` indicates how a row is evaluated under a given valuation `v` of variables $x_1, ..., x_n$. The function `dot_prod` implements the usual dot product on vectors of the same size: the resulting integer is the sum of every member of the valuation multiplied by the corresponding coefficient. Given a valuation, `eval_SPV` evaluates an SPV into the PI of a single word. Function `lan_SPV` specifies that the language recognised by an SPV is the image of the evaluation function.

```
Definition row n := vec ℕ n * ℕ.
Definition SPV n := Map.t (row n).

Definition val n := vec ℕ n.
Definition eval_row (v: val n) (r: row n) := dot_prod v (fst r) + snd r.
Definition eval_SPV (v: val n) (p: SPV n) : PI Σ := λx ⇒ eval_row v p[x].
Definition lan_SPV (p: SPV n) : lan := λpi ⇒ ∃val, pi ≡ₙ eval_SPV val p.
```

Listing 4 Symbolic Parikh Vectors.

We implement the transformation from expressions into SPVs as a function `dnf_to_SPVs` which we do not detail here. The correctness of this function is expressed by the lemma below, which states that a word is recognised by an expression e if and only if it is recognised by one of the SPVs generated from e. Note that this lemma only applies if e is a `dnf`: the normalisation algorithm must be applied before transformation into SPVs.

▶ **Lemma 3.** Translation into SPVs preserves the recognised language

```
∀ e, dnf e → ∀pi, pi ∈ lan e ↔ ∃p, p ∈ dnf_to_SPVs e ∧ pi ∈ lan_SPV p
```

3.3 Generating Linear Diophantine Equations

In the previous subsections, we have shown how to convert commutative regular expressions into disjunctions of SPVs. We now build systems of linear Diophantine equations that represent the intersection of a pair of SPVs. A linear Diophantine equation is a linear equation with integer coefficients and solutions. As a first step, our algorithm represents these equations with natural coefficients, and allow coefficients and constants on both sides of the equations (e.g. $2x_1 + 3x_2 + 1 = 4x_3$). The second step translates these equations into a stricter form compatible with our solver.

We describe this process through the example from Figure 1. To calculate the intersection of expressions $(aab)^*b^*$ and $(abb)^*a$, the algorithm first calculates their two SPVs

$$\{a \mapsto 2x_1; b \mapsto x_1 + x_2\} \qquad \text{and} \qquad \{a \mapsto x_3 + 1; b \mapsto 2x_3\}$$

Recall that variables x_1, x_2, x_3 represent the number of repetitions of each * in the source expressions. We denote the number of occurrences of a and b in a given word by x_a and x_b, respectively. This word belongs to the intersection of the two SPVs whenever there exists an assignment to x_1, x_2, x_3 satisfying the following system (system S_1 in Figure 1):

$$\begin{cases} -\ 0c + 1x_a + 0x_b - 2x_1 - 0x_2 - 0x_3 = 0 \\ -\ 1c + 1x_a + 0x_b - 0x_1 - 0x_2 - 1x_3 = 0 \\ -\ 0c + 0x_a + 1x_b - 1x_1 - 1x_2 - 0x_3 = 0 \\ -\ 0c + 0x_a + 1x_b - 0x_1 - 0x_2 - 2x_3 = 0 \end{cases}$$

Each equation corresponds to the association between one letter and one expression in one of the SPVs (e.g. the second equation corresponds to $a \mapsto x_3 + 1$ from the second SPV). The first column corresponds to the number of occurrences of the letter in the constant term, i.e. not under a star. For these equations to accurately describe the intersection, c must be equal to 1: the constant term is counted exactly once. Adding this "dummy" variable allows us to represent a non-homogeneous system as a homogeneous one, as done in [10, Sec 7].

Listing 5 presents our implementation of systems of Linear Diophantine Equations. Type `diophs n m` represents a system of m Diophantine equations with n variables as a matrix of coefficients of m vectors of size n. The language recognised by such a system is defined by `lan_dioph`. It depends on a vector of letters `v` which gives the correspondence between columns of the system and letters of the language. A PI `pi` is recognised if (1) every letter in `pi` appears in `v`, and (2) there exists a solution to the system of equations in which the variable corresponding to each letter `x` is instantiated to `pi[x]` (`vmap` is the usual map function

```
Definition dioph n := vec ℤ n.
Definition diophs n m := vec (dioph n) m.

Definition eval_dioph (d: dioph n) (v: vec ℕ n) := dot_prod d v.
Definition solution (d: diophs n m) (v: vec ℕ n) :=
  ∀ r, r ∈ d ⇒ eval_dioph r v = 0.
Definition lan_dioph (v: vec Σ n) (d: diophs (1 + n + n2) _) : lang :=
  λ pi ⇒ (∀ x, x ∈ pi → x ∈ v) ∧ ∃v2, solution d (vcons 1 (vmap pi v ++ v2)).
```

Listing 5 Representation and evaluation of systems of linear Diophantine equations.

on vectors). To evaluate whether or not a valuation `v` is a solution of the system, every equation is evaluated independently by taking the dot product of its coefficients with `v`.

Going back to the intersection algorithm, a function `SPVs_to_diophs` (not detailed here) converts a pair of SPVs into the system of equations denoting their intersection. It returns the vector of letters and the matrix of coefficients. Lemma 4 specifies its correctness.

▶ **Lemma 4.** *The language recognised by the system of equations computed from a pair of SPVs is exactly the intersection of the languages of the two SPVs*

```
∀ s1 s2, let '(v, deqs) := SPVs_to_diophs s1 s2 in
lan_dioph v deqs ≡ lan_spv s1 ∩ lan_spv s2
```

Chaining the transformations presented in this section, we get a function that translates two expressions e_1 and e_2 into two normalised expressions (e.g., $(aab)^*b^*$ and $(aab)^*a+(aab)^*b$ from Figure 1), which are each translated into a list of SPVs (a list containing $\{a \mapsto 2x_1; b \mapsto x_1 + x_2\}$ and another with $\{a \mapsto x_3 + 1; b \mapsto 2x_3\}$ and $\{a \mapsto x_4; b \mapsto 2x_4 + 1\}$). Then, it generates a system of equations for each pair of SPVs in the cartesian product of these lists (systems S_1 and S_2). By chaining the previous correctness lemmas, the union of the languages recognised by these systems of equations is equal to `lan` e_1 ∩ `lan` e_2.

4 Formally Verified Solving of Diophantine Equation Systems

We now move on to the second half of our intersection algorithm, which solves the systems of equations generated in the first part, and rebuilds a commutative regular expression from the solution. To do so, we implemented the algorithm proposed by Contejean and Devie [10] as a pure Rocq function. In the first part of this section, we give a high-level overview of the algorithm through an iterative characterisation that allowed us to easily mechanise its proof of correctness. Then, in Section 4.2, we describe the executable implementation of the algorithm, focusing on its proof of termination. In Section 4.3, we show a more practical formulation of the correctness of this algorithm. Finally, in Section 4.4, we show how the solution is used to rebuild a commutative regular expression representing the intersection.

4.1 Iterative Characterisation and Correctness

The goal of the algorithm proposed by Contejean and Devie is to compute a set of *minimal* solutions to a system of linear Diophantine equations. We first recall the definition of a minimal solution, which we mechanised as presented in Listing 6. In this section, we assume that the system of equations `eqs` is fixed. A vector of natural numbers is a minimal solution to this system if (1) it is a solution, (2) it is non-null (characterised by its L1-norm $|\cdot|$ being strictly positive), and (3) there exists no solution that is strictly smaller according to the partial ordering `vgt` (noted `>>` in [10]): `v >> u` iff each component of `v` is greater than or equal to the corresponding one in `u`, and at least one component of `v` is strictly greater than the corresponding one in `u`. For instance, `(1,2,3)>>(1,1,3)`.

We now focus on the implementation of the simplest version of the algorithm, presented in [10, Sec 4], which computes the set of minimal solutions. This algorithm uses a while loop to iterate until a set of candidate solutions is empty. It is not possible to implement this algorithm as-is in Rocq, because all recursive Rocq functions must be proven to be terminating. The criteria for termination is that every recursive call must be *guarded*: one argument of the function must be inductive, and every recursive call must apply to a subterm of this argument (we say that this argument is *strictly decreasing*). In the case of this

```
Variable eqs : diophs n m.

Definition vgt (u v: vec ℕ n) := (∀ i, u_i ≥ v_i) ∧ (∃ i, u_i > v_i).

Definition min_solution (v: vec ℕ n) :=
  solution eqs v ∧ |v| > 0 ∧ (∀ x, solution eqs x → |x| > 0 → ¬(v >> x)).
```

Listing 6 Minimal solutions to a system of equations.

algorithm, there is no good candidate for a strictly decreasing argument. Indeed, as the authors state, the proof that this algorithm terminates is non-trivial.

As a first approximation, and to facilitate the proof of correctness, we defined a recursive function `iter` that uses a natural number k as a "fuel" argument to iterate the body of the while loop k times. This function is presented in Listing 7. The `step` function corresponds to one step of the loop. It takes two lists of vectors as inputs. List `b` contains minimal solutions that were already found. List `p` contains solution *candidates*: a vector `x` is a candidate if there is no minimal solution `s` in `b` such that `x >> s`. The function transforms `p` and `b`. First, it extracts actual solutions from `p` and adds them to `b`. Then, for every vector `x` left in `p`, it tries to add to `x` any unit vector $\hat{e}_i$ that satisfies Fortenbacher's restriction [6]: $eqs(x) \cdot eqs(\hat{e}_i) < 0$. This restriction "cuts" the hyperplan perpendicularly to $eqs(x)$, keeping $eqs(x + \hat{e}_i)$ in the half-space containing the origin. This restriction reduces the search space, and, in particular, is crucial to guaranteeing termination by preventing the algorithm from continuing its search "infinitely far away" from the solution. We implement this check by taking the `dot_prod` of `eval x` and `eval` $\hat{e}_i$. Finally, `p` is filtered to keep only the vectors for which all previously computed solutions in `b` are not strictly greater. As said, the `iter` function iterates `step` k times, starting with an empty set of solutions, and the unit vectors for candidates.

```
Definition eval (v: vec ℕ n) : vec ℤ m := vmap (λ d → eval_dioph d v) eqs.

Definition step (p b: list (vec ℕ n)) : pair (list (vec ℕ n)) :=
  let (bn, p) := partition solution? p in
  let b := b ∪ bn in
  let p :=
    ⋃x ∈ p
      (let evx := eval x in
       let fbasen := filter (λ ei ⇒ dot_prod evx (eval ei) <? 0) [ê1;...;ên] in
       let p := map (λ ei ⇒ x + ei) fbasen in
       filter (λ v ⇒ forallb (λ s ⇒ negb (v >> s)) b) p)
  in (p, b).

Fixpoint iter (k: ℕ) : pair (list (vec ℕ n)) :=
  match k with
  | 0 ⇒ ([ê1;...;ên], [])
  | S k ⇒ let (p, b) := iter k in step p b
  end.
```

Listing 7 Iterative characterisation of the system solver.

In the following, suppose `iter k = (p, b)`. Proving that `iter` is correct amounts to proving that it returns a vector `v` in `b` if and only if `v` is a minimal solution. Let us start with the proof of completeness: all minimal solutions are computed. For a given `k`, we cannot prove

that this implementation computes all minimal solutions, since it may stop before finding all of them. In fact, the algorithm will find only (but all) minimal solution `v` such that `|v|` $\leq$ `k`: this is formalised by the lemma below. The proof of this lemma relies on two intuitions. First, any solution in `p` is found and added to `b`. Second, any solution with norm greater than `k` is reachable from `p`. This second point is the most difficult to prove, because it requires proving that Fortenbacher's restriction does not suppress some of the solutions. The proof proposed by Contejean and Devie consists in showing that any solution may be decomposed in a sum of unit vectors, all leading in the right direction. We mechanised this argument into an inductive predicate, `decomposable: vec ℕ n → ℙ`, showed that `min_solution v` implies `decomposable v`, and finished the proof by induction on the derivation of this predicate.

▶ **Lemma 5.** *Completeness (Proposition 1 from [10])*

```
∀ v k, let (p, b) := iter k in min_solution v → |v| <= k → v ∈ b
```

The second part of the correctness proof is soundness: any vector `v` in `b` is actually a minimal solution. It is easy to see that `v` is a solution (since this is checked by `filter`), and that it is positive (since all vectors in `p` are greater than or equal to a unit vector). To prove that it is minimal, we reason by contradiction: if it is not, then there exists a minimal solution `x` such that `v >> x`. By the completeness lemma, we know that this solution was found by `iter k'` with `k' < k`. Because of the filtering applied at the end of the `step` function, that would mean `v` could not be in `p`, and therefore not in `b`, which is a contradiction.

▶ **Lemma 6.** *Soundness (Proposition 2 from [10])*

```
∀ v k, let (p, b) := iter k in v ∈ b → min_solution v
```

4.2 Implementation and Termination

In this section we describe our formalisation of Contejean and Devie's termination argument. To describe this mechanisation as a proof of termination for a Rocq function, let us start from the implementation presented in Listing 8. The conclusion of the argument is that, at some point, the list of candidate solutions p will be empty. Therefore, the recursive function `solve_aux` first checks if p (the first element of the pair `pb`) is empty. If it is, it returns b (the second element of the pair). If it is not, it calls itself recursively after one call to `step`. This recursive call is not guarded a priori, and Rocq would normally refuse this function. However, we follow the approach recommended by Leroy [20], and add an extra argument to the function, `ACC`, which acts as the decreasing argument. `ACC` is a proof of *accessibility* for the relational form of the step function, `StepR`: it states that any sequence of elements related by `StepR` starting from `pb` is finite. Inverting `ACC` requires proving that the `step` function satisfies its relational specification, which is easily done in `StepR_step`. This is sufficient to define a recursive function which is accepted by Rocq.

Of course, this is not the whole proof: by using `Acc StepR`, we have merely moved the burden of the proof to the initial caller of `solve_aux`, the `solve_diophs` function, which must prove that any sequence starting from the initial list of candidates is finite, which corresponds exactly to the argument of Contejean and Devie, and is formalised in lemma `StepR_init`.

To mechanise this argument, we proceeded, as in the original paper, by contradiction. The central piece of reasoning is Lemma 7 presented below, which corresponds to [10, Prop 3]. Let `VR v1 v2` be the relation between a candidate solution `v2` and `v1`, its predecessor by the `step` function. Let `v` be an infinite sequence of candidate solutions. We write `v k` for the k-th element of `v`, which is a vector in $\mathbb{N}^n$. If `v` respects `VR`, then there exists at least one index `k`

```
Definition StepR pb1 pb2 := fst pb1 <> [] ∧ step_rel pb1 pb2.

Lemma StepR_step : ∀p b, p <> [] → StepR (p, b) (step p b).
Proof. (* easy *) Qed.

Fixpoint solve_aux pb (ACC: Acc StepR pb) :=
  match list_empty_dec (fst pb) with
  | left _ ⇒ snd pb
  | right NEMPTY ⇒
      solve_aux (step (fst pb) (snd pb)) (Acc_inv ACC (StepR_step pb NEMPTY))
  end.

Lemma StepR_init : Acc StepR ([ê₁;...;êₙ], []).
Proof. (* hard, uses proposition 3 *) Qed.

Definition solve_diophs := solve_aux ([ê₁;...;êₙ], []) StepR_init.
```

Listing 8 Recursive implementation of the system solver.

and one positive solution `x` to the system of equations, such that `v k` is strictly greater than `x`. This is a contradiction: thanks to the reasoning used in proving Lemma 6, we know that any candidate solution must not be strictly greater than a minimal solution.

▶ **Lemma 7.** *There is no infinite sequence respecting `StepR` (Proposition 3 from [10])*

```
let VR v1 v2 := ∃j, v2 = v1 + ê_j ∧ dot_prod (eval v1) (eval ê_j) < 0 in
∀ (v: ℕ → vec ℕ n),
  (∃ j, v 0 = ê_j ∧ ∀k, VR (v k) (v (S k))) →
  (∃ k x, solution eqs x ∧ |x| > 0 ∧ (v k) >> x)
```

As Contejean and Devie state, the proof of Lemma 7 is "quite delicate". Indeed, it involves techniques and results from real analysis and linear algebra. The proof is (summarily) structured as follows:

1. First, it shows that the sequence $\frac{||\texttt{eval (v k)}||}{\texttt{k}}$, with $||.||$ denoting the L2-norm, converges to 0 ([10, Lemma 1]). The intuition is that $||\texttt{eval (v k)}||$ grows slower than `k`, since the difference between `(v (k+1))` and `(v k)` is a unit vector. Using submultiplicativity of the Frobenius norm, we conclude that $||\texttt{eval}\ \frac{\texttt{(v k)}}{\texttt{k}}||$ also converges to 0.
2. Then, since $\frac{\texttt{(v k)}}{\texttt{k}}$ is bounded, it admits an adherence value $l \in \mathbb{R}^n$. This vector is proven to have at least one strictly positive component, since it is the adherence value of a sequence of non-null vectors $\frac{\texttt{(v k)}}{\texttt{k}}$. Moreover, it is a solution for `eqs` in $\mathbb{R}^n$, since $||\texttt{eval}\ \frac{\texttt{(v k)}}{\texttt{k}}||$ converges to 0.
3. l is decomposed into a vector l_1 with only strictly positive components and a null vector. In the mechanisation, we use explicit projection functions between l and l_1. l_1 is further decomposed into a vector l_2 of components that are linearly independent in $\mathbb{Q}$, and a second vector l_3 where every component is a sum of components of l_2 coefficiented by $\mathbb{Q}$.
4. From there, it extracts a solution to the system of equations $w \in \mathbb{Q}^n$. By multiplying every component in w, it gets back a strictly positive solution $v_0 \in \mathbb{N}^n$.
5. v_0 is then proven to be strictly smaller than some `v k`, which is a contradiction: this particular `v k` cannot have been generated by the algorithm. This finished the proof.

To mechanise this proof, we used the `Reals` module provided with Rocq's standard library. We needed to generalize some results from this library; for instance, the Bolzano-Weierstrass

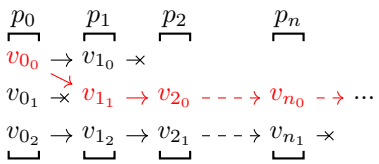

Figure 2 Choosing vectors.

▶ **Lemma 8.** *Functional Backwards List Choice*

```
∀ (R: A → A → ℙ) (p : ℕ → list A),
(∀ k, p k <> []) →
(∀ k v, v ∈ (p (k+1)) → ∃u, u ∈ (p k) ∧ R u v) →
∃ v, In (v 0) (p 0) ∧ ∀k, R (v k) (v (k+1))
```

theorem which was only defined for scalar reals and needed to be generalized to vectors of reals. We also needed to add some definitions to manipulate vectors by index during the proof, in particular when extracting components of l and constructing w. Overall, the proof of Lemma 7 is the most complex in the development, and requires approximately 2,000 LoC.

We also discovered a missing conceptual step in the pen-and-paper proof proposed by Contejean and Devie. Let us go back to `StepR_init`: we must prove that any sequence of pairs (p, b) satisfying `StepR` is finite. We proceed by contradiction: we must prove that the existence of an infinite sequence of pairs (p, b) satisfying `StepR` is a contradiction. Let us only consider p, the list containing the candidate solutions, for each pair. Figure 2 illustrates the infinite sequence of p, where arrows represent the `VR` relation. We know that there is an infinite number of non-empty p_n, and that every element in p_{n+1} is related to an element in p_n. From this assumption, we need to extract an infinite sequence of vectors related by `VR`, to instantiate Lemma 7 and finish the proof. The corresponding proof obligation is a specialisation of Lemma 8, with `A := vec ℕ` and `R := VR`. However, we realised that it is not possible to prove this lemma constructively. Indeed, it is not possible to write a terminating algorithm that chooses between a sequence that is actually infinite (such as $v_{0_0}, v_{1_1}, v_{2_0}, \ldots$ in the example), and a sequence that is finite but may be arbitrarily long (such as the one starting at v_{0_2} in the example). This is akin to the halting problem. Instead, the proof relies on non-constructive reasoning, in particular with the axioms of excluded-middle and of dependent functional choice, and is similar to the proof of König's infinity lemma [16].

4.3 Correctness theorem

In the previous sections, we have established that `solve_diophs` terminates, and proved the correctness of its iterative specification. The correctness of `solve_diophs` is obtained by combining this result with a proof of correspondence between the recursive implementation and the iterative specification. This correctness is stated below, in terms of minimal solutions.

▶ **Lemma 9.** *Correctness of the Diophantine equations solver in terms of minimal solutions*

```
∀ eqs v, min_solution eqs v ↔ v ∈ (solve_diophs eqs)
```

However, to simplify further reasoning, we state the final correctness theorem in terms of *linear combinations* of vectors. The inductive predicate `lin_comb vs v` presented below specifies that `v` is a linear combination of vectors in `vs` with coefficients in ℕ.

```
Inductive lin_comb: list (vec ℕ n) → vec ℕ n → ℙ :=
| lin_comb_nil: lin_comb [] 0⃗
| lin_comb_cons: ∀hd tl x v1, lin_comb tl v1 → lin_comb (hd::tl) (x * hd + v1).
```

Listing 9 Linear combinations of vectors.

Intuitively, any vector is a solution to the system of equations iff it is equal to a linear

combination of minimal solutions returned by `solve_diophs`, as specified below.

▶ **Theorem 10.** *Correctness of the Diophantine equations solver*

```
∀ eqs v, solution eqs v ↔ lin_comb (solve_diophs eqs) v
```

4.4 From the Set of Minimal Solutions to Expression

Finally, after having computed the set of minimal solutions to the system of equations generated for an intersection, it remains to generate a commutative regular expression from this solution. Recall that all equations in the system have the same structure, with the first variable corresponding to the number of occurrences of the constant term, the n_1 next variables to the number of occurrences of each letter in the intersection, and the remaining n_2 variables being free. Naturally, the solution vectors also follow this structure. As explained in Section 3.3, we are only interested in solutions where the first component is exactly 1. Since every solution to the system can be expressed as a linear combination of minimal solutions, we immediately exclude minimal solutions whose first component is greater than 1, as proposed in [10, Sec 7]. The first subset, m_1, contains "constant" solutions. The second subset, m_0, contains "repeating" solutions. Any solution of interest is thus the sum of exactly one vector from m_1 and of repeated vectors from m_0.

As an example, the system of equations presented on Page 7 for the first intersection in Figure 1 has the following set of minimal solutions:

$$\{(0\ 2\ 4\ 1\ 3\ 2), (1\ 2\ 2\ 1\ 1\ 1), (3\ 4\ 2\ 2\ 0\ 1)\}$$

Any solution of the system is thus of the form $(0\ 2\ 4\ 1\ 3\ 2) \times k + (1\ 2\ 2\ 1\ 1\ 1) \times 1$ for some $k \in \mathbb{N}$. Focusing on variables x_a and x_b (second and third values), in all solution we have $x_a = 2k + 2$ and $x_b = 4k + 2$, for some k. Since x_a and x_b express the number of a's and b's, respectively, in the intersection, the SPV of the expression denoting the intersection is $\{a \mapsto 2k + 2; b \mapsto 4k + 2\}$. Applying the inverse transformation from Section 3.2, we obtain the resulting expression as follows. The repeated parts of a and b are bounded by the same variable (k) and so they appear under the same starred word, a twice and b four times. Since k is the only variable, there are no other starred words, only constant occurrences of a and b, twice each. The resulting expression is therefore $(aabbbb)^*aabb$.

This transformation is implemented by the functions presented in Listing 10. The function `split_sol` splits the solution into the two subsets m_0 and m_1, and removes the first component of each vector to simplify later processing. Each vector in the resulting subsets is converted into a product expression by the function `vec_to_exp`, which is parameterised by the vector of letters computed earlier by `diophs_of_eqs`. With a vector of letters $(x_1, ..., x_{n_1})$ and a solution vector $(k_1, ..., k_{n_1}, k_{n_1+1}, ..., k_{n_1+n_2})$, the letter x_i is repeated k_i times in the generated expression. The components after k_{n_1} are ignored, as they correspond to free variables. Finally, the function `sol_to_exp` combines these functions to transform the whole set of minimal solutions. First, it decomposes this set into the two subsets `m1` and `m0`. Then, it generates a sum of expressions from the vectors in `m1` and a product of starred expressions from the vectors in `m0`.

The correctness lemma for `sol_to_exp` is presented below. It establishes that the generated expression preserves the language recognised by the system of equations. This lemma holds for any input set of vectors that forms a minimal solution, in the sense of Theorem 10.

```
Definition split_sol (m: list (vec ℕ (1 + n))) :=
  (map vtail (filter (λ v ⇒ vhead v =? 1) m),
   map vtail (filter (λ v ⇒ vhead v =? 0) m)).

Fixpoint vec_to_exp (vn: vec Σ n1) (v: vec ℕ (n1 + n2)) :=
  match vn, v with
  | vnil, vnil ⇒ One
  | vcons x vn', vcons k v' ⇒ Prod (⨉_k x) (vec_to_exp vn' v')
  end.

Definition sol_to_exp (vn: vec Σ n1) (m: list (vec ℕ (1 + n1 + n2))) :=
  let '(m1, m0) := split_sol m in
  Prod (⨁_{v ∈ m1} (vec_to_exp vn v)) (⨉_{v ∈ m0} (Star (vec_to_exp vn v))).
```

Listing 10 Translating a solution into an expression.

▶ **Lemma 11.** *The expression computed by* `sol_to_exp` *corresponds to the set of minimal solutions of the equations*

```
∀ eqs sol,
  (∀ v, solution eqs v ↔ lin_comb sol v) →
  lan (sol_to_exp vn sol) ≡ lan_dioph names eqs
```

4.5 Putting it all together

Recall that computing the intersection of two expressions required solving a disjunction of systems of Diophantine equations. Lastly, the expressions generated from the solutions of these systems are combined using the sum operator. Listing 11 presents the final intersection function, built from composing the functions presented in the two previous sections.

```
Definition intersection (e1 e2: exp) : exp :=
  let pvs1 := dnf_to_SPVs (normalise e1) in
  let pvs2 := dnf_to_SPVs (normalise e2) in
  ⨁_{p1 ∈ pvs1, p2 ∈ pvs2}
    (let '(vn, deqs) := SPVs_to_diophs p1 p2 in sol_to_exp vn (solve_diophs deqs))
```

Listing 11 Intersection of commutative regular expressions.

The final correctness theorem for this function is presented below, and is proved by composing the correctness lemmas for each of these functions.

▶ **Theorem 12.** *Correctness of the intersection*

```
∀ e1 e2, lan (intersection e1 e2) ≡ lan e1 ∩ lan e2
```

5 Performance and Experimental Evaluation

In their paper, Contejean and Devie propose several versions of the intersection algorithm. The Rocq function described in the previous section is an implementation of the "naïve", high-level version of the algorithm. While testing, we realised that this implementation could be quite slow on some relatively simple input expressions: for example, computing the

intersection of $((a^2b^3 + a^5)bc^2)^*$ and $(a^5b^2)^*c^*$—which is $(a^{90}b^{36}c^{42})^*$—takes 5 seconds on a laptop with an Intel Ultra 7 running at 4.5GHz. Most of this time is consumed by the equation-solving algorithm (normalisation takes less than 0.0001 seconds). This is due to the solutions being the fairly large vector $(90, 36, 42)$, which requires at least $90 + 36 + 42 = 168$ iterations to be found by the solving algorithm.

5.1 A more efficient algorithm

To improve on these performances, we implemented in Rocq the optimised incremental stack algorithm [10, Sec 6&9], which makes several improvements over the naïve version. First, it is incremental, meaning that it solves the system one equation at a time, refining its minimal solution with each new equation. It starts with the canonical base, which is the minimal solution of an empty set of equations. Then, when adding another equation to the system, the algorithm searches a new minimal solution as a set of linear combinations of vectors from the previous minimal solution. This approach reduces the number of iterations of the search, since every step of the search adds a vector from the base computed from the previous equations rather than a unit vector. Second, the algorithm fixes the order in which the vectors of the basis may be added. This avoids redundancies in the search: if the algorithm computed $v_1 + v_2$, it will never compute $v_2 + v_1$. Third, it avoids explicit computation of the `<<` operator by instead keeping track of the difference between the candidate and solution vectors. Last, the search uses a stack, on which vectors are pushed and popped in an order that guarantees that the size of the stack is at most the number of vectors in the previous minimal solution, making the algorithm space-efficient. For the example above, this algorithm computes the same intersection in under 0.001 seconds.

While implementing this version of the algorithm, we detected two small errors in its original pseudo-code [10, page 160]: an incorrect decrement when calculating the height of the stack at the start of each loop cycle (`n:=height(P)-1`), and the vector comparison with existing solutions in the wrong direction (`>>` instead of `<<`). At time of writing, the termination and correctness proofs for this version of the algorithm are still work in progress. The optimisations introduced in the stack algorithm make reasoning much more complex; in particular, it requires complex invariants relating the previous minimal solution, the order in which vectors from it have been added into candidate vectors on the stack, and the computed differences. We are confident the proofs are doable given enough time.

5.2 Simplifying expressions

Expressions produced by the normalisation process and by the conversion of solutions to expressions may contain redundant terms. This has two drawbacks. First, the intermediate expressions may be translated to more complex unions of SPVs than necessary, and therefore more complex systems of equations, impacting the solving time. Second, the output expressions may be less readable. To alleviate these issues, we added an expression simplifier in our pipeline. It is applied after normalisation, and before returning the output expression. It simplifies useless 1 and 0 (e.g. $e1 \equiv e$ and $e + 0 \equiv 0$), redundancies in sums (e.g. $e + e \equiv e$) and in product of starred words (e.g. $e^*e^* \equiv e^*$). The simplifier also has been fully verified.

5.3 Evaluation

To evaluate the performance of our implementation, we generated a benchmark with several batches of uniformly randomly generated regular expressions using the tool FAdo [2, 29].

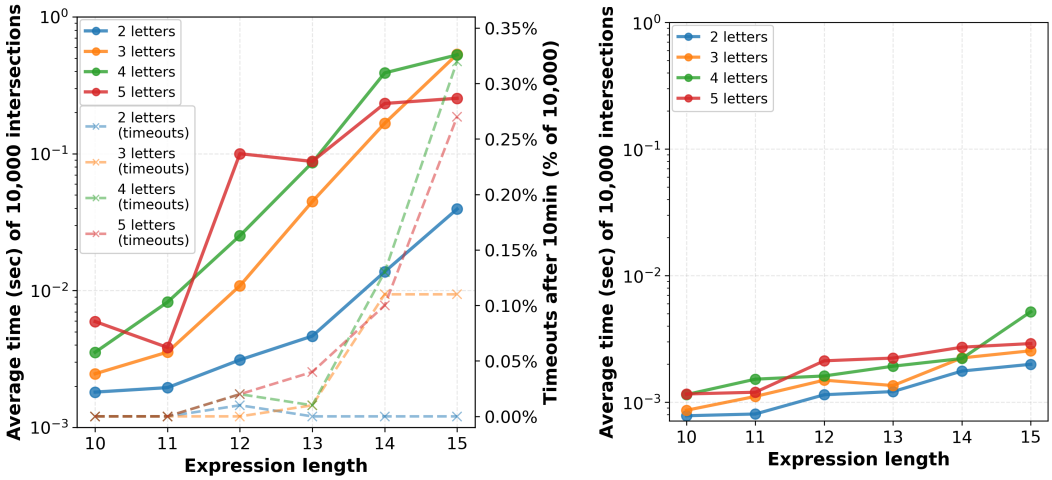

Figure 3 Average time taken to compute intersections of random expressions using two versions of the Contejean-Devie algorithm: fully formalised original version (left) vs partially formalised stack incrementer version (right).

Each batch contains 10,001 expressions of a fixed length (varying between 10 and 15) and a fixed number of letters (between 2 and 5) in Polish notation (so their lengths are statistically significant). Then, for every two consecutive expressions in a batch we computed their intersection and measured the time taken. The results are presented in Figure 3, which includes the results obtained with the more efficient version described in Section 5.1 (right chart) for comparison and as motivation for its ongoing formalisation. For each combination of expression length and number of letters, we present 1) the average time of the intersections that finished in 10 minutes (continuous lines), and 2) the percentage of intersections that did time out (dashed lines). This illustrates a wide amplitude in the computation times. Despite this, we observed average and median times consistently below 1 second. Expression length clearly is a factor, although some of the times actually dropped when length increased.

To better understand the test cases that timed out, we analysed more closely the first 5 cases that did not finish within 10 minutes for expressions of size 15 and 4 letters. With the stack algorithm all 5 intersections were computed in ≤ 0.10 seconds. Two of these intersections were as simple as $(ddcb)^*$ and $(dddcaaaab)^*$, but our certified implementation, with timeout disabled, took ~ 8 hours to compute the first one. The second one was still running after a full day. This can happen when the equation solver quickly finds relatively few minimal solutions but then spends most of the time exploring all remaining candidate solutions until discarding them (i.e., each of them keeps being incremented repeatedly without ever becoming a solution, but eventually becoming strictly greater than an already-found minimal solution). For two of the remaining cases, the intersections computed were significantly larger, one containing 52 terms and taking ~ 6 hours to compute, and the other 7 terms but taking a full day. For the last one, the stack algorithm computed an intersection with 168 terms but the certified implementation was still running after 1 day.

The stack algorithm exhibits a more consistent behaviour, without timeout. Apart from the expression length, the number of letters also has an influence, with, for instance, expressions with only 2 letters consistently intersecting faster than the other ones. This is unsurprising, as the number of equations in the system is double that of the letters.

We also tested our implementation with real-world examples from mailbox types [15],

which are defined as commutative regular expressions. For instance, we can compute the intersections of the types inferred by the MBCheck typechecker for the Pat language [14]. During the typechecking of the Dining Philosophers implementation included in the tool, two of the types inferred are

`(H(D+1))*H(DE+1)` and `(H(D+1))*H(DE+1)(H(D+1))*H(DE+1)+(H+D+E)*`

Where `H`, `D` and `E` are short for `Hungry`, `Done` and `Exit`, respectively. Their intersection is computed in under 0.01s by our certified implementation:

`H*(DH)*DEH+H*(DH)*DEHH+H*(DH)*H+H*(DH)*HH`.

6 Discussion and Future Work

We closed the open problem of how to compute the intersection of two commutative regular expressions. While commutative regular expressions were already known to be closed under intersection, no algorithm existed in the literature yet. We introduced an algorithm, formalised it in Rocq and proved it correct. Since our algorithm relies on solving systems of linear Diophantine equations, our mechanisation includes a formalisation of such a solver (in this case, the solver proposed by Contejean and Devie [10]). While solvers for *single* linear Diophantine equations had already been formalised before [22], this is the first formalisation of a solver for *systems* of linear Diophantine equations, which works both with homogeneous and non-homogeneous systems. Both the algorithm that computes expression intersections and the solver of systems of linear Diophantine equations are available to use in the executable OCaml-extracted artefact provided [3]. The artefact also includes the first benchmark of intersections of commutative regular expressions, computed from uniformly random generated regular expressions using our certified implementation.

We estimate that 3 researcher-months were required to complete the formalisation work described in this paper. The result of this work is around 5.5k lines of Rocq code, including specification, executable code and proofs for every pass of the algorithm.

Table 1 Number of LoC and axioms used for each part of the mechanisation

	Lines of Code				
Pass	Specification	Executable	Proof	Total	Axioms Used
Exp + Axioms	60	0	570	630	*none*
Normalisation	15	100	540	655	`classic`
Exp → SPV	15	50	340	405	*none*
SPV → Eqs	15	25	230	270	*none*
Eqs Solver	10	30	3100	3140	`classic`, `DFC`, real number axioms
Solution → Exp	0	20	420	440	*none*
Total	115	225	5200	5540	

As shown in Table 1, the bulk of the development lies in the proof of correctness for the Contejean-Devie algorithm presented in Section 4. This was expected, considering the pen-and-paper proof was already "quite delicate" and spanned more than 4 pages in the original paper. Our mechanisation follows this proof closely, while clarifying some of the steps: for instance, the "continuity arguments" used on pages 11 and 12 of the original paper respectively involve the submultiplicative property of the Frobenius norm, and the density of $\mathbb{Q}$ in $\mathbb{R}$.

As discussed in the previous sections, the proofs of termination and correctness of the Diophantine solver are not constructive. More generally, our development relies on five non-constructive axioms from Rocq's standard library, which may be listed using the `Print Assumptions` command. They can be broadly categorised into:

- Excluded-middle (`classic`) and dependent functional choice (`DFC`): necessary to prove Lemma 8, similarly to the proof of König's infinity lemma.
- Axioms used to define real numbers in Rocq's library (`sig_not_dec`, `sig_forall_dec` and `funext`): necessary to prove the termination of the Contejean-Devie algorithm.

The last column of Table 1 recaps which axioms are used in which part of the proof.

Pottier [27] has provided a bound on the norm of the minimal solutions that could be used as an alternative when proving the algorithm terminates. Contejean and Devie point out that this bound is incomparable to Fortenbacher's restriction, which we implemented, but report that it yielded a larger set of nodes in all their experiments [10], while additionally being significantly more complex to compute. This is the main reason why we chose not to use it, but it would still be interesting to implement it and test how it affects the performance of the solver in practice. It may also allow us to remove some of the non-constructive axioms used in the termination proof.

In addition to being a (semi-)constructive proof that commutative regular expressions are closed under intersection, the algorithm we implemented may be used in practical contexts, such as type inference for process calculi (e.g. Mailbox types [15]). One limitation is that the algorithm cannot be run inside Rocq itself, since the decreasing argument of the `solve_aux` function presented in Listing 8 is a proof of accessibility which contains applications of non-constructive axioms, and therefore does not reduce. Instead, we use the Rocq extraction feature to generate equivalent OCaml code, which we then compile and run.

In Section 2.1, we introduced Redko's theorem, that states that every identity between commutative regular expressions is a consequence of the axioms. A possible direction of future work is to extend our formalisation (which already includes all axioms) with an algorithm that decides the equivalence between any two expressions. As in the case of the intersection, we expect this work would reuse some of the ideas and concepts from the existing non-constructive proofs (such as Conway's, which refines the normal form into a new *independent* form [11]).

Finally, we plan to integrate the executable code extracted from our formalisation into the OCaml tool BigraphER [31]. As mentioned in the introduction, bigraph sorts are given by commutative regular expressions and an implementation of the intersection will enable the first implementation of sort inference for bigraphs [4].

Declaration on the use of AI

We declare that, for the experimental evaluation described in Section 5, the parser for input expressions and the Python script to plot our results (Figure 3) were written with the aid of GenAI.

References

1 Farid Ajili and Evelyne Contejean. Avoiding slack variables in the solving of linear diophantine equations and inequations. *Theoretical Computer Science*, 173(1):183–208, 1997. doi:10.1016/S0304-3975(96)00195-8.

2 André Almeida, Marco Almeida, José Alves, Nelma Moreira, and Rogério Reis. FAdo and GUItar: Tools for automata manipulation and visualization. In Sebastian Maneth, editor,

Implementation and Application of Automata, pages 65–74, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. doi:10.1007/978-3-642-02979-0_10.

3 Ricardo Almeida, Blair Archibald, Basile Pesin, and Michele Sevegnani. Certified intersection of commutative regular expressions as solutions of systems of linear diophantine equations, May 2026. doi:10.5281/zenodo.20265929.

4 Blair Archibald and Michele Sevegnani. A Bigraphs Paper of Sorts. In Russ Harmer and Jens Kosiol, editors, *Graph Transformation*, pages 21–38. Springer Nature Switzerland, 2024. doi:10.1007/978-3-031-64285-2_2.

5 Iovka Boneva, Radu Ciucanu, and Sławek Staworko. Schemas for unordered xml on a dime. *Theory of Computing Systems*, 57(2):337 – 376, 2015. doi:10.1007/s00224-014-9593-1.

6 Michael Clausen and Albrecht Fortenbacher. Efficient solution of linear diophantine equations. *Journal of Symbolic Computation*, 8(1):201–216, 1989. doi:10.1016/S0747-7171(89)80025-2.

7 Manuel Clavel, Francisco Durán, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. Unification and narrowing in maude 2.4. In Ralf Treinen, editor, *Rewriting Techniques and Applications*, pages 380–390, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. doi:10.1007/978-3-642-02348-4_27.

8 Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All about maude-a high-performance logical framework: how to specify, program, and verify systems in rewriting logic*, volume 4350. Springer, 2007. doi:10.1007/978-3-540-71999-1.

9 Evelyne Contejean. Solving *-problems modulo distributivity by a reduction to ac1-unification. *Journal of Symbolic Computation*, 16(5):493–521, 1993. doi:10.1006/jsco.1993.1060.

10 Evelyne Contejean and Hervé Devie. An Efficient Incremental Algorithm for Solving Systems of Linear Diophantine Equations. *Information and Computation*, 113(1):143–172, 1994. doi:10.1006/inco.1994.1067.

11 John H. Conway. Regular Algebra and Finite Machines. 1971. URL: https://api.semanticscholar.org/CorpusID:117721644, doi:10.2307/3616008.

12 Justin DeBenedetto and David Chiang. Algorithms and training for weighted multiset automata and regular expressions. In Cezar Câmpeanu, editor, *Implementation and Application of Automata*, pages 146–158, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-94812-6_13.

13 Ugo de'Liguoro and Luca Padovani. Mailbox Types for Unordered Interactions. In Todd Millstein, editor, *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, volume 109 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:28, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2018.15.

14 Simon Fowler. MBCheck: A typechecker for the Pat language. URL: https://github.com/SimonJF/mbcheck.

15 Simon Fowler, Duncan Paul Attard, Franciszek Sowul, Simon J. Gay, and Phil Trinder. Special Delivery: Programming with Mailbox Types. 7:78–107, 2023. doi:10.1145/3607832.

16 Miriam Franchella. On the origins of dénes könig's infinity lemma. *Archive for History of Exact Sciences*, 51(1):3–27, Mar 1997. doi:10.1007/BF00376449.

17 Alexander Herold and Jörg H Siekmann. Unification in abelian semigroups. *Journal of Automated Reasoning*, 3(3):247–283, 1987. doi:10.1007/BF00243791.

18 John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Introduction to automata theory, languages, and computation, 2nd edition. *SIGACT News*, 32(1):60–65, March 2001. doi:10.1145/568438.568455.

19 Eryk Kopczynski. Complexity of problems of commutative grammars. *Logical Methods in Computer Science*, Volume 11, Issue 1, Mar 2015. doi:10.2168/LMCS-11(1:9)2015.

20 Xavier Leroy. Well-founded recursion done right. In *CoqPL 2024: The Tenth International Workshop on Coq for Programming Languages*, London, United Kingdom, January 2024. ACM. URL: https://inria.hal.science/hal-04356563.

21 GS Makanin. Algorithmic decidability of the rank of constant free equations in a free semigroup. In *Dokl. Akad. Nauk, SSSR*, volume 243, 1978.

22 Florian Meßner, Julian Parsert, Jonas Schöpf, and Christian Sternagel. A formally verified solver for homogeneous linear diophantine equations. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving*, pages 441–458, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-94821-8_26.

23 Robin Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009. doi:10.1017/CBO9780511626661.

24 Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989. doi:10.1109/5.24143.

25 Dávid Papp and Béla Vizvári. Effective solution of linear diophantine equation systems with an application in chemistry. *Journal of Mathematical Chemistry*, 39:15–31, 01 2006. doi:10.1007/s10910-005-9001-9.

26 Donald Pilling. *The Algebra of Operators for Regular Events*. University of Cambridge, 1970. URL: https://books.google.co.uk/books?id=vPB40AEACAAJ.

27 Loïc Pottier. Minimal solutions of linear diophantine systems: bounds and algorithms. In *Proceedings of the 4th International Conference on Rewriting Techniques and Applications*, RTA-91, page 162–173, Berlin, Heidelberg, 1991. Springer-Verlag. doi:10.1007/3-540-53904-2_94.

28 V.N. Redko. On the algebra of commutative events. *Ukrainian Math. J.*, 16(16):185 – 195, 1964. In Russian. URL: https://www.scopus.com/pages/publications/0000257607.

29 Rogério Reis and Nelma Moreira. FAdo 2.2.0, 2024. URL: https://fado.dcc.fc.up.pt/.

30 Arto Salomaa. *Theory of automata*. Elsevier, 2014. doi:10.1016/C2013-0-02221-9.

31 Michele Sevegnani and Muffy Calder. BigraphER: Rewriting and analysis engine for bigraphs. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 494–501. Springer, 2016. doi:10.1007/978-3-319-41540-6_27.

32 Slawek Staworko, Iovka Boneva, Jose E. Labra Gayo, Samuel Hym, Eric G. Prud'hommeaux, and Harold Solbrig. Complexity and Expressiveness of ShEx for RDF. In Marcelo Arenas and Martín Ugarte, editors, *18th International Conference on Database Theory (ICDT 2015)*, volume 31 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 195–211, Dagstuhl, Germany, 2015. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ICDT.2015.195.

33 Mark E. Stickel. A unification algorithm for associative-commutative functions. *J. ACM*, 28(3):423–434, July 1981. doi:10.1145/322261.322262.