EPiC Series in Computing

Volume 57, 2018, Pages 164–180

LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning

A Verified Efficient Implementation of the LLL Basis Reduction Algorithm

Ralph Bottesch, Max W. Haslbeck, and René Thiemann

University of Innsbruck, Austria

Abstract

The LLL basis reduction algorithm was the first polynomial-time algorithm to compute a reduced basis of a given lattice, and hence also a short vector in the lattice. It thereby approximately solves an NP-hard problem. The algorithm has several applications in number theory, computer algebra and cryptography.

Recently, the first mechanized soundness proof of the LLL algorithm has been developed in Isabelle/HOL. However, this proof did not include a formal statement of the algorithm's complexity. Furthermore, the resulting implementation was inefficient in practice.

We address both of these shortcomings in this paper. First, we prove the correctness of a more efficient implementation of the LLL algorithm that uses only integer computations. Second, we formally prove statements on the polynomial running-time.

1 Introduction

The LLL basis reduction algorithm, originally introduced by (and named after) Lenstra, Lenstra and Lovász [11], is a remarkable algorithm with numerous applications. The algorithm computes an approximate solution to the following problem:

Shortest Vector Problem (SVP): Given a linearly independent set of m vectors, $f_0, \ldots, f_{m-1} \in \mathbb{Z}^n$, which form a basis of the corresponding *lattice* (the set of vectors that can be written as linear combinations of the f_i, with integer coefficients), compute a non-zero lattice vector that has the smallest-possible norm.

This problem plays an important role in number theory and cryptography [14]. It is NP-hard to solve exactly in general [13], but, given any basis of a lattice L as input, the LLL algorithm computes, in polynomial time, a basis of L that is *reduced w.r.t.* α, which implies, among other things, that the shortest vector in the basis is at most $\alpha^{\frac{m-1}{2}}$ times larger than the shortest non-zero vector in the lattice. Here, $\alpha > \frac{4}{3}$ is a parameter of the algorithm that also appears in the running time.

In recent work, Divasón, Joosten, Thiemann, and Yamada [5] developed the first mechanized proof of the soundness of the LLL algorithm, using Isabelle/HOL [17]. Since Isabelle code can be exported to other programming languages and then run on actual data, their work results in a verified implementation of the LLL algorithm. Having verified implementations of algorithms

is important not mainly because the correctness of the algorithms themselves might be in doubt, but because such implementations can be composed into large reliable programs, of which every part has been formally proved to work as intended.

Our first contribution is to modify the verified implementation of the LLL algorithm from [5] so as to make it considerably faster. Although both the input and the output of instances of SVP are sets of integer-valued vectors, the original formalization followed a particular textbook version of the algorithm, that makes extensive use of computations on rational numbers. We determined via tests that gcd computations, which are necessary in order to reduce fractions, accounted for at least 83 % of the running time of that implementation on each input. In order to improve on this, we followed [7, 18] to obtain a fully verified, integer-only implementation of the LLL algorithm, thus eliminating the need for the use of rational numbers altogether.

The corresponding generated Haskell code now runs in time comparable to that of the LLL implementation in some commercial software packages for mathematical computations, such as Mathematica (see Figure 1). Specifically, in numerical experiments, our new implementation was, at worst, about 7x slower than Mathematica; usually the two were much closer in speed. This means that, in addition to having the advantage of being formally verified, our implementation is now usable in practice. By contrast, on lattices of dimension $n \geq 30$, the old verified implementation is at least $3n$ times slower than the new one. However, it should be noted that specialized floating-point implementations of LLL like fplll [19] are still orders of magnitude faster than either our new implementation or the one in Mathematica.

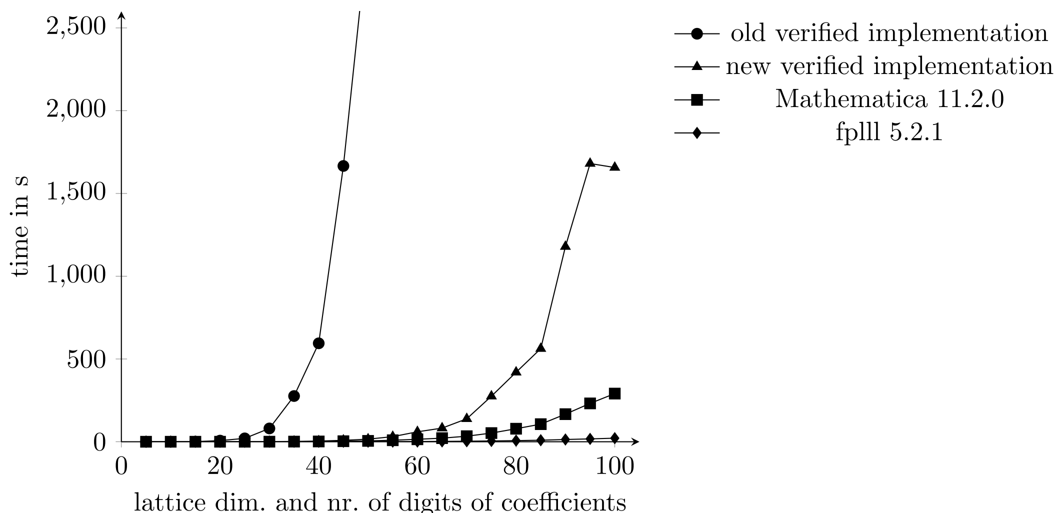

Figure 1: A comparison of the performance of LLL implementations on lattices obtained from instances of polynomial factorization. The verified implementations were run with $\alpha = \frac{3}{2}$. Mathematica and fplll used their default values for α, which are even closer to $\frac{4}{3}$, resulting in slightly better approximations of shortest vectors.

Our second contribution is a formal proof of a polynomial bound on the running-time of the rational version of the algorithm from [5] (for which only the correctness, not the complexity bound, was formally proved), as well as the running time of the new integer version. We focus mainly on the latter in our presentation, but in both cases, the complexity bound is proved by first showing a polynomial bound on the number of arithmetic operations, and then showing that throughout the entire execution of the algorithm, the computed numbers can be represented

using a number of bits that is polynomial in the size of the input. The two main Isabelle lemmas expressing these facts are the following:

- **lemma** *reduce_basis_cost_expanded:*
  assumes $Lg = nat \lceil\ log\ (of_rat\ (4 \cdot \alpha\ /\ (4 + \alpha)))\ A\ \rceil$
  and $A = Max\ \{\|v\|^2 \mid v.\ v \in set\ fs\}$
  shows $cost\ (reduce_basis_cost\ fs) \leq 49 \cdot m^3 \cdot n \cdot Lg$ `(* illustrative bound *)`

  The function *reduce_basis_cost* is an extended version of *reduce_basis* (which implements the (integer) LLL algorithm). The extended function returns the result of the original function, together with the number of arithmetic operations required to compute it. The above lemma then gives a polynomial upper bound on this number of required operations, where A is the maximum squared norm of the input vectors, and Lg is the logarithm of A with base $\frac{4\alpha}{4+\alpha}$ (which is > 1 when $\alpha > \frac{4}{3}$). The above bound is a simplified version of the finer bound from the code, and only serves to illustrate the fact that it is polynomial.

- **lemma** *combined_size_bound_integer:* **assumes** ...
  and $M = Max\ \{|fs\ !\ i\ \$\ j| \mid i\ j.\ i < m \wedge j < n\}$
  and $x \in ...$ `(* description of numbers during run of algorithm *)`
  shows $log\ 2\ |x| \leq (6 + 6 \cdot m) \cdot log\ 2\ (M \cdot n) + m + log\ 2\ m$

  The second lemma combines the size bounds for all numbers computed by the algorithm throughout its run. Here, M is the maximum absolute value occurring in the input fs.

Together, the two lemmas imply the polynomial complexity of our implementation of the LLL algorithm, since each arithmetic operation can be computed in polynomial time, and only polynomially many such operations are executed.

The two contributions amount to a considerable expansion of the original project of [5], with the code base having roughly doubled. The new proofs are available in the Archive of Formal Proofs (AFP) for Isabelle 2018, entry `LLL_Basis_Reduction` [1]. All definitions and lemmas found in this paper are also links which lead to an HTML version of the corresponding Isabelle theory file. The code referenced here can be found in the following theories: `Gram_Schmidt_Int.thy`, `LLL_Number_Bounds.thy`, `LLL_Integer_Equations.thy`, `LLL_Mu_Integer_Impl.thy`, and `LLL_Mu_Integer_Impl_Complexity.thy`. We provide further installation instructions for the formalization at http://cl-informatik.uibk.ac.at/isafor/experiments/lll. This website also contains experimental data, such as the input matrices, the experimental setup, and the Haskell code of the old and the new verified LLL implementation.

We briefly discuss how the present work ties in with other related projects. As an example of verified software we mention CeTA [3, 20], a tool for checking untrusted termination proofs and complexity proofs. One of the checks performed by this software requires computations with algebraic numbers. Although verified implementations of algebraic numbers are already available both in Coq [2] and in Isabelle/HOL [12, 22], there is still room for improvement: since the algebraic number computations heavily rely upon polynomial factorization, the verification of a fast factorization algorithm would greatly improve the performance of these implementations. A natural choice would then be van Hoeij's algorithm [24], which is currently the fastest deterministic polynomial factorization algorithm (it successfully factors polynomials in seconds on examples where a verified implementation [4] does not finish even in hours). Since van Hoeij's algorithm uses the LLL basis reduction algorithm as a subroutine, a future verified version of it can make full use of the efficiency of our new LLL implementation.

The remaining sections are organized as follows: We first recall the existing formalization in Section 2. In Section 3 we discuss the details of implementing and proving the correctness of the integer LLL algorithm in Isabelle. We illustrate the formal proof of the bound on the number of arithmetic operations in Section 4, and present the bounds on the size of the numbers in Section 5. We discuss the technicalities of using Cramer's lemma in Isabelle in Section 6. Finally, we conclude in Section 7.

2 Preliminaries

Interactive theorem proving

Our tool of choice for the formalization of proofs is Isabelle/HOL. We assume familiarity with it, and refer the reader to [16] for a quick introduction, but nevertheless we briefly review some Isabelle notation, which should make most of the code segments accessible to readers familiar only with standard mathematical notation.

All terms in Isabelle must have a well-defined type, specified with a double-colon: *term :: type*. Type variables have a ′ sign before the identifier. The type of a function with domain *A* and range *B* is specified as *A* ⇒ *B*. Each of the base types *nat*, *int* and *rat* corresponds to the number set suggested by its name. Access to an element of a vector, list, or array is denoted, respectively, by *$*, *!*, *!!*. For example, if *fs* is of type *int vec list*, the type of lists of vectors of integers, then *fs ! i $ j* denotes the j-th component of the i-th vector in the list. In the text, we will often use more common mathematical notation instead of Isabelle notation. For example, we would write f_i rather than *fs ! i*. The syntax for function application in Isabelle is *func arg1 arg2 ...*; terms are separated by white spaces, and *func* can be either the name of a function or a lambda expression. Note that some terms that we index with subscripts in the in-text mathematical notation are defined as functions in the Isabelle code (for example $\mu_{i,j}$ stands for *mu i j*).

The Formalized LLL Algorithm

In this section we briefly review the existing Isabelle/HOL formalization of the LLL algorithm [5], focusing only on the process of formally verifying its correctness; for an explanation of the algorithm itself, we refer to [25] or [14].

The algorithm to be formalized is given as pseudo-code in Algorithm 1. Here, $\lfloor x \rceil = \lfloor x + \frac{1}{2} \rfloor$ is the integer nearest to x, the inner product of vectors u and v is denoted by $u \bullet v$, and $\|u\|^2 = u \bullet u$ is the squared Euclidean norm of u. The values g_i and $\mu_{i,j}$ are defined as follows, with f always referring to the current values of f in Algorithm 1:

$$g_i = f_i - \sum_{j<i} \mu_{i,j} \cdot g_j \qquad\qquad \mu_{i,j} := \begin{cases} 1 & \text{if } j = i \\ 0 & \text{if } j > i \\ \frac{f_i \bullet g_j}{\|g_j\|^2} & \text{if } j < i. \end{cases}$$

The vectors g are the so-called *Gram–Schmidt orthogonalization (GSO) vectors* and the recursive definitions of g and μ describe the *Gram–Schmidt orthogonalization procedure*. If $f_0, \ldots, f_{m-1}$ are a list of linearly independent vectors in $\mathbb{R}^n$ or $\mathbb{Q}^n$, then $g_0, \ldots, g_{m-1}$ are an orthogonal basis for the space spanned by $f_0, \ldots, f_{m-1}$. This procedure has already been formalized in Isabelle as a function *gram_schmidt* in the proof of existence of the Jordan normal forms [23].

Algorithm 1: The LLL basis reduction algorithm, verified version

Input: A list of linearly independent vectors $f_0, \ldots, f_{m-1} \in \mathbb{Z}^n$ and $\alpha > \frac{4}{3}$
Output: A basis for the same lattice as $f_0, \ldots, f_{m-1}$, that is reduced w.r.t. α

```
1 i := 0
2 while i < m do
3     for j = i − 1, . . . , 0 do
4         f_i := f_i − ⌊μ_{i,j}⌉ · f_j
5     if i > 0 ∧ ‖g_{i−1}‖^2 > α · ‖g_i‖^2 then
6         (i, f_{i−1}, f_i) := (i − 1, f_i, f_{i−1})
      else
7         i := i + 1
8 return f_0, . . . , f_{m−1}
```

The overall approach to formalizing the rest of Algorithm 1 is as follows (although we note that the presentation here hides some refinements). First, the algorithm is encoded in several functions (see the explanations following the code). From here onward, throughout the rest of the paper, we present Isabelle code residing in a context that fixes the approximation factor α, the dimensions n and m, and the basis fs_{init} of the initial (input) lattice.

```
definition basis_reduction_step :: (nat × int vec list) ⇒ (nat × int vec list) where
  basis_reduction_step (i, fs) = ...                    (* implementation of lines 3–7 *)

partial_function (tailrec) basis_reduction_main :: (nat × int vec list) ⇒ int vec list where
  basis_reduction_main (i, fs) = (
    if i < m
    then basis_reduction_main (basis_reduction_step (i, fs))
    else fs)

definition reduce_basis :: int vec list ⇒ int vec list where
  reduce_basis fs = basis_reduction_main (0, fs)
```

Some remarks about the above code fragments:

- The body of the while-loop (lines 3–7) is modeled by the function *basis_reduction_step*, the details of which we omit.
- The while-loop itself (line 2) is modeled as the partial function *basis_reduction_main*. Note that the function is not necessarily terminating, since there is no restriction on the validity of inputs (e.g. $\alpha = 0$ is not ruled out). Putting these assumptions into the context might be possible for proving properties of the LLL algorithm, but would prevent code-generation.
- Finally, the full algorithm is implemented as the function *reduce_basis*, which starts the loop and then returns the final integer basis $f_0, \ldots, f_{m-1}$.

Next, in order to prove the correctness of the algorithm, an *invariant* is defined, which is simply a set of conditions that the current state must satisfy throughout the entire execution of the algorithm. In the following definition, *lin_indpt_list* is a predicate expressing linear

independence. The predicate *reduced k fs* requires that the current g and μ values of the first *k* vectors of *fs* are *reduced w.r.t.* α, i.e., that $\mu_{i,j} \leq \frac{1}{2}$ holds for all $j < i < k$ and that $\|g_i\|^2 \leq \alpha \cdot \|g_{i+1}\|^2$ holds for all $i < k - 1$.

definition *LLL_invariant i fs* = (
  lin_indpt_list fs ∧
  lattice_of fs = *lattice_of fs_init* ∧
  length fs = *m* ∧
  reduced i fs ∧
  i ≤ *m*)

The key correctness property of the LLL algorithm is then given by the following lemma, which states that the invariant is preserved in the while-loop of Algorithm 1 (specifically, that if the current state, prior to the execution of an instruction, satisfies the invariant, then so does the resulting state after the instruction). The lemma also states that a measure indicating how far the algorithm is from completing the computation, is decreasing – this is used to prove that the algorithm terminates.

lemma *basis_reduction_step:*
  assumes *LLL_invariant i fs* **and** *i* < *m*
    and *basis_reduction_step* (*i*, *fs*) = (*i′*, *fs′*)
  shows *LLL_invariant i′ fs′*
    and *LLL_measure i′ fs′* < *LLL_measure i fs*

Finally, using Lemma *basis_reduction_step*, one can prove the following crucial properties of the LLL algorithm. Again, *A* is the maximum squared norm of the initial lattice basis fs_{init}.

1. The resulting basis is reduced and is a basis for the same lattice as the initial basis. The first element of the reduced basis is an approximation of the shortest vector in the lattice.

   lemma *short_vector:*
     assumes *v* = *hd* (*reduce_basis fs*) **and** *h* ∈ *lattice_of fs* − {*0*}
     shows $\|v\|^2 \leq \alpha^{m-1}\|h\|^2$

2. The algorithm terminates, since the *LLL_measure* is decreasing in each iteration.

3. The number of loop iterations is bounded by *LLL_measure i fs* when invoking the algorithm on inputs *i* and *fs*, so *reduce_basis* requires at most *LLL_measure i fs* many iterations.

4. *LLL_measure i fs* $\leq m + 2 \cdot m \cdot m \cdot$ *log* $(\frac{4 \cdot \alpha}{4+\alpha})$ *A*

These properties have all been stated and proved in [5]. There, the verified algorithm already contains some optimizations, e.g., the g vectors are incrementally updated whenever f is changed, instead of computing g from scratch in every iteration. Using the upper bound for the *LLL_measure*, one can derive a total bound of $\mathcal{O}(m^3 \cdot n \cdot \log A)$ arithmetic operations for the LLL algorithm. However, this has not been done formally in [5], nor has a bound on the values of f_i, g_i and $\mu_{i,j}$ been formally proved. Especially, the latter property is not at all obvious and it is easily violated by small changes in the algorithm, cf. the last paragraphs of [5, Section 4].

3 A Formally Verified Integer Implementation of the LLL Algorithm

In this section we describe the formalization in Isabelle of a version of the LLL algorithm that uses only integers. As mentioned in the introduction, such an implementation is desirable for efficiency reasons.

To obtain an implementation of the LLL algorithm that uses only integer operations, we modify the Isabelle implementation of this algorithm from [5]. This modification is necessary because in order to perform the essential computations of the LLL algorithm, one needs to keep track of either the μ-matrix or the GSO-vectors corresponding to the (current) set of f vectors, neither of which contain integers in general. We therefore replace the μ-matrix by a matrix of integers with a similar meaning.

Given a set of vectors $v_1, \ldots, v_n$, entry (i, j) of the corresponding *Gramian matrix* is defined to be $v_i \bullet v_j$. In our case, the vectors v_i are the f_i. The *Gramian determinant* then, is the determinant of the Gramian matrix.

There are multiple ways to define and characterize the Gramian matrix and determinant in Isabelle. From here onward, all of the code we show resides in a context in which the f vectors form a linearly independent set.

definition *Gramian_matrix fs k* = (**let** *M* = *mat k n* (λ(*i, j*). (*fs ! i*) *$ j*) **in** *M* $\cdot$ *M*^{T})

lemma assumes *k* < *m*
shows *Gramian_matrix fs k* = *mat k k* (λ(*i, j*). *fs ! i* $\bullet$ *fs ! j*)

For brevity of notation, we will denote *Gramian_determinant fs k* by d_k or *d k*, unless we wish to emphasize that d_k is defined as a determinant. Note that *Gramian_determinant* depends explicitly on *fs*, whereas this dependency is implicit for *d*; whenever a quantity depends implicitly on *fs*, we assume that we are working in a context where *fs* is fixed.

definition *d k* = *det* (*Gramian_matrix fs k*)

lemma *Gramian_determinant:*
assumes *k* ≤ *m*
shows *d k* = ($\prod$ *j*<*k*. $\|$*gs ! j*$\|^2$)

Apart from the integer implementation, the Gramian determinant also appears when proving the termination of the LLL algorithm (it is used to define *LLL_measure*), as well as when proving upper bounds on the numbers that are computed in the course of a run of the algorithm (see Section 5).

The most important fact for the integer implementation is given by the following lemma:

lemma *Gramian_determinant_mu_ints:*
assumes *j* ≤ *i* **and** *i* < *m*
shows *d* (*Suc j*) $\cdot$ μ *i j* $\in \mathbb{Z}$

Based on this fact we derive a LLL implementation which only tracks the values of $\tilde{\mu}$, where $\tilde{\mu}_{i,j} := d_{j+1}\mu_{i,j}$ (in the Isabelle source code, $\tilde{\mu}$ is called *d*μ). We can show that the $\tilde{\mu}$ values can be calculated using only integer arithmetic, and that it suffices to keep track of only these values in the LLL algorithm.

3.1 An Integer Implementation of Gram–Schmidt Orthogonalization

Since the LLL algorithm performs Gram–Schmidt orthogonalization as a subroutine, a full integer-only implementation of the former also requires such an implementation of the latter. For this, we mainly follow [7], where a GSO-algorithm using only operations in an abstract integral domain is given. We made some small simplifications to this algorithm and then formalized in Isabelle most of the proofs from [7].

Algorithm 2: Gram–Schmidt orthogonalization (adapted from [7]) – for $\tilde{\mu}$-values only

Input: A list of linearly independent vectors $f_0, \ldots, f_{m-1} \in \mathbb{Z}^n$
Output: $\tilde{\mu}$ where $\tilde{\mu}_{i,j} = d_{j+1}\mu_{i,j}$

```
1 for i = 0,...,m−1 do
2     μ̃_{i,0} := f_i • f_0
3     for j = 1,...,i do
4         σ := μ̃_{i,0} μ̃_{j,0}
5         for l = 1,...,j−1 do
6             σ := (μ̃_{l,l} σ + μ̃_{i,l} μ̃_{j,l}) div μ̃_{l−1,l−1}
7         μ̃_{i,j} := μ̃_{j−1,j−1} (f_i • f_j) − σ
8 return μ̃
```

The correctness of Algorithm 2 hinges on two properties: that the calculated $\tilde{\mu}_{i,j}$ are equal to $d_{j+1}\mu_{i,j}$, and that it is correct to use integer division in line 6 of the algorithm (in other words, that the intermediate values computed at every step of the algorithm are integers). We prove these two statements in Isabelle by starting out with a more abstract version of the algorithm, which we then refine to the one above. Specifically, we first define the relevant quantities

definition $\tilde{\mu}$ *i j* = *d* (*Suc i*) $\cdot$ μ *i j*
fun σ **where**
  σ *0 i j* = *0*
| σ (*Suc l*) *i j* = (*d* (*Suc l*) $\cdot$ σ *l i j* + $\tilde{\mu}$ *i l* $\cdot$ $\tilde{\mu}$ *j l*) / *d l*

where σ *l i j* represents the value of σ at the *l*-th pass through the innermost loop. Note that the type of (the range of) $\tilde{\mu}$ and σ is *rat* and not *int*, which is why we can use general division (for fields) in the above function definition, rather than *div*. The advantage of letting $\tilde{\mu}$ and σ have more general types is that we can proceed to prove all of the equations and lemmas from [7] while focusing only on underlying mathematics, without having to worry on non-exact division. For example, from the definition above we can easily show the following characterization:

lemma σ*:* **assumes** $l \leq m$
  shows σ *l i j* = *d l* $\cdot$ ($\sum$ *k* < *l*. μ *i k* $\cdot$ μ *j k* $\cdot$ $\|gs\ !\ j\|^2$)

which is needed to prove one of the two statements that are crucial for the correctness of the algorithm:

lemma σ*_integer:*
  assumes $l \leq j$ **and** $j \leq i$ **and** $i < m$
  shows σ *l i j* $\in \mathbb{Z}$

We also mention that the proof of Lemma *σ_integer* requires an application of Cramer's lemma. The difficulties with applying this lemma in Isabelle are described in Section 6.

The other ingredient required to prove correctness is to show how $\tilde{\mu}$ can be computed in terms of σ.

lemma $\tilde{\mu}$: **assumes** $j \le i$ **and** $i < m$
shows $\tilde{\mu}\ i\ j = d\ j \cdot (fs\ !\ i \cdot fs\ !\ j) - \sigma\ j\ i\ j$

Having proved the desired properties of the abstract version of the algorithm, we make the connection with an actual implementation that computes the values of $\tilde{\mu}$ recursively. Here, the identities $d_{j+1} = \tilde{\mu}_{j,j}$ and $d_0 = 1$ are also included; they show that the $\tilde{\mu}$-values include the d-values in particular.

```
fun σℤ :: nat ⇒ nat ⇒ nat ⇒ int and μ̃ℤ :: nat ⇒ nat ⇒ int where
  σℤ 0       i j = μ̃ℤ i 0 · μ̃ℤ j 0
| σℤ (Suc l) i j = (μ̃ℤ (Suc l) (Suc l) · σℤ l i j
                     + μ̃ℤ i (Suc l) · μ̃ℤ j (Suc l)) div μ̃ℤ l l
| μ̃ℤ i j = (if j = 0 then fs ! i · fs ! j
                else μ̃ℤ (j − 1) (j − 1) · (fs ! i · fs ! j) − σℤ (j − 1) i j)
```

Note that these functions only use integer arithmetic and therefore return a value of type *int*. We then show that the new functions are equal to the ones defined previously. Here, *of_int* is a function that converts a number of type *int* into the corresponding number of type *rat*. Further note that the indices of $\sigma_{\mathbb{Z}}$ are shifted by 1 with respect to the indices of σ. This is for the sake of ease of implementation.

lemma $\sigma_{\mathbb{Z}}$_$\tilde{\mu}$: $l < j \Longrightarrow j \le i \Longrightarrow i < m \Longrightarrow of_int\ (\sigma_{\mathbb{Z}}\ l\ i\ j) = \sigma\ (Suc\ l)\ i\ j$
$i < m \Longrightarrow j \le i \Longrightarrow of_int\ (\tilde{\mu}_{\mathbb{Z}}\ i\ j) = \tilde{\mu}\ i\ j$

We then replace the repeated calls of $\tilde{\mu}_{\mathbb{Z}}$ by saving already computed values in an array for fast access. Furthermore, we rewrite $\sigma_{\mathbb{Z}}$ to be a tail-recursive function, which completes the integer implementation of the algorithm.

Note that Algorithm 2 so far only computes the $\tilde{\mu}$-matrix. This in particular includes the d_i values, since $d_{i+1} = d_{i+1} \cdot 1 = d_{i+1} \cdot \mu_{i,i} = \tilde{\mu}_{i,i}$ and $d_0 = 1$.

For completeness, we also formalized Algorithm 3, which computes integer-valued multiples of the GSO-vectors. This, in turn, required us to also formally prove that all of the intermediate values (specifically, the values of τ in each iteration) are integer vectors, so that the vector-by-scalar division div_v is exact division in each invocation. We again prove the correctness of this algorithm by first defining an abstract version and then refining it to an optimized and executable version.

3.2 An Integer Implementation of the LLL Algorithm

Recall that the verified implementation of Algorithm 1 in [5] stores both f and g, but recomputes the μ-values on the fly. Alternatively, one can store the f vectors, the μ values, and the norms of the g vectors, in which case the g vectors themselves are no longer required [11]. The latter approach has the advantage that from this representation it is easy to switch to an implementation that only stores f, the $\tilde{\mu}$-matrix, and the d-values, which, by Lemma $\sigma_{\mathbb{Z}}$_$\tilde{\mu}$, are all integer values and integer vectors [18]. This integer representation will be the basis for

Algorithm 3: Gram–Schmidt orthogonalization (adapted from [7]) – $\tilde{g}$ vectors only

Input: A list of linearly independent vectors $f_0, \ldots, f_{m-1} \in \mathbb{Z}^n$ and $\tilde{\mu}$
Output: $\tilde{g}$ where $\tilde{g}_i = d_i g_i$

1 compute $\tilde{\mu}$ by Algorithm 2
2 $\tilde{g}_0 := f_0$
3 **for** $i = 1, \ldots, m-1$ **do**
4 $\quad \tau := \tilde{\mu}_{0,0} f_i - \tilde{\mu}_{i,0} f_0$
5 $\quad$ **for** $l = 1, \ldots, i$ **do**
6 $\quad\quad \tau := (\tilde{\mu}_{l,l}\tau - \tilde{\mu}_{i,l}\tilde{g}_l) \operatorname{div}_v \tilde{\mu}_{l-1,l-1}$
7 $\quad \tilde{g}_i := \tau$
8 **return** $\tilde{g}$

our verified integer implementation of the LLL algorithm. To prove its soundness, we proceed similarly as for the GSO procedure: We first provide an implementation which still operates on rational numbers and uses field-division. We then use Lemma $\sigma_{\mathbb{Z}}$-$\tilde{\mu}$ to implement and prove soundness of an equivalent but efficient algorithm which only operates on integers.

First, we need to extend the soundness properties of the existing verified LLL algorithm. For instance, Lemma *basis_reduction_step* in Section 2 only speaks about the effect w.r.t. the invariant, of executing one while-loop iteration of Algorithm 1, but it does *not* provide results on how to update the $\tilde{\mu}$-values and the d-values. To this end, we added several *computation* lemmas of the following form, which precisely specify how the values of interest are updated when performing a swap of f_i and f_{i-1}, or when performing an update $f_i := f_i - c \cdot f_j$. As in Section 2, the newly computed values of *gs*, *d*, and μ, are marked with a $'$ sign after the identifier.

lemma *basis_reduction_add_row_main:* **assumes**...
  and $fs' = fs\ [i := fs\ !\ i - c \cdot fs\ !\ j]$ (* operation on f *)
  and $j < i$ **and** $i < m$
shows...
  and $k < m \Longrightarrow gs'\ k = gs\ k$ (* no change in GSO *)
  and $k \leq m \Longrightarrow d'\ k = d\ k$ (* no change in d−values *)
  and $i_0 < m \Longrightarrow j_0 < m \Longrightarrow \mu'\ i_0\ j_0 =$ (* change of μ *)
    (**if** $i_0 = i \wedge j_0 \leq j$ **then** $\mu\ i_0\ j_0 - c \cdot \mu\ j\ j_0$ **else** $\mu\ i_0\ j_0$)

The computation lemma above is more versatile than just proving that the LLL-invariant is maintained in step 3 of Algorithm 1. The precise description of the μ-values allows us to establish the invariant easily: if $c = \lfloor \mu_{i,j} \rceil$, then the new $\mu_{i,j}$-value will be small afterwards and only the μ_{i,j_0}-entries with $j_0 \leq j$ can change. Moreover, the computation lemma allows us to implement this part of the algorithm for various representations, i.e., one obtains local updates for f, g, μ, and d.

Whereas the above computation lemmas mainly speak about rational numbers and vectors, we further provide similar computation lemmas for the integer values and vectors f, $\tilde{\mu}$, and d, in such a way that the new values can completely be calculated based on the previous integer values of f, $\tilde{\mu}$, and d. At this point, we also replace field divisions by integer divisions; the corresponding soundness proofs heavily rely upon Lemma $\sigma_{\mathbb{Z}}$-$\tilde{\mu}$. As an example, the computation lemma for the swap operation of f_{k-1} and f_k provides the following equality for d and a more

complex one for the update of $\tilde{\mu}$.[1]

$d'\ i = (\textbf{if}\ i = k\ \textbf{then}\ (d\ (Suc\ k) \cdot d\ (k - 1) + (\tilde{\mu}\ k\ (k - 1))^2)\ div\ d\ k\ \textbf{else}\ d\ i)$

After having proved all the updates for $\tilde{\mu}$ and d when changing f, it remains to implement all the other expressions in Algorithm 1 based on these integer values. For instance, the expression $\|g_{i-1}\|^2 \leq \alpha\|g_i\|^2$ is shown to be equivalent to $d_i^2 \cdot denom \leq num \cdot d_{i-1} \cdot d_{i+1}$ where α is represented by its numerator *num* and denominator *denom*. Similarly, $\lfloor \mu_{i,j} \rceil$ is implemented as $(2 \cdot \tilde{\mu}_{i,j} + d_{j+1})\ \mathsf{div}\ (2 \cdot d_{j+1})$.

Finally, we plug everything together to obtain an executable LLL algorithm that uses solely integer operations. It has the same structure as Algorithm 1 and therefore we are able to prove that the integer algorithm and Algorithm 1 behave alike regarding the changes to the f vectors, only the internal representations being different. Consequently, we just reuse the existing soundness lemmas for Algorithm 1 like Lemma *basis_reduction_step*, in order to prove soundness of our integer implementation of the LLL algorithm.

We end this section with a small discussion on the general principles underlying our formalization. One can clearly identify an instance of a refinement approach: we conclude soundness of the integer algorithm via the soundness of the rational number algorithm in combination with a refinement relation (e.g., Lemma $\sigma_{\mathbb{Z}}$_$\tilde{\mu}$) that connects both algorithms. The formulation of computation rules might be applicable for other algorithm as well. However, we do not see how to generalize the reasoning on why certain values during the execution of this specific algorithm are integers.

4 A Formally Verified Bound on the Number of Arithmetic Operations

In this section we provide details on how the Lemma *reduce_basis_cost_expanded* from the introduction was proved. The first step to be able to reason about the number of arithmetic operations is to extend the whole algorithm by annotating and collecting costs. In our cost model, we only count the number of arithmetic operations.

To integrate this model formally, we use the same lightweight approach as in [6]. It has the advantage that it was very easy to integrate on top of the existing formalization.

- We use a type *'a cost* = *'a* × *nat* to represent a result of type *'a* in combination with a cost for computing the result.
- For every Isabelle function *f* :: *'a* ⇒ *'b* that is used to define the LLL algorithm, we define a corresponding extended function *f_cost* :: *'a* ⇒ *'b cost*. These extended functions use pattern matching to access the costs of sub-algorithms, and then return a pair where all costs are summed up.
- In order to state correctness, we define two selectors *cost* :: *'a cost* ⇒ *nat* and *result* :: *'a cost* ⇒ *'a*. Then soundness of *f_cost* is split into two properties. The first one states that the result is correct: *result* (*f_cost x*) = *f x*, and the second one provides a cost bound *cost* (*f_cost x*) ≤ . . ..

[1]The updates for $\tilde{\mu}_{i,j}$ consider five different cases depending on the relations between i, j, and k.

We did not use ressource monads as in [15] – which can be used to accumulate the costs – to model the functions *f_cost*. The reason is that we would then always have to break the monad abstraction in order to formally prove the cost bounds.

We illustrate our approach using two example functions: *dmu_array_row_main_cost* corresponds to lines 3–7 of Algorithm 2, and *basis_reduction_main_cost* is an annotated version of *basis_reduction_main* as it is defined in Section 2.

```
function dmu_array_row_main_cost where
  dmu_array_row_main_cost fi i dmus j = (let ...
     (σ, c1) = sigma_cost ...                          (* c1: cost of computing σ *)
     dmu_ij = djj · (fi • fs !! sj) − σ                (* 2n + 2 arith. operations *)
     dmus' = iarray_update dmus i j dmu_ij             (* array update, no cost *)
     (res, c2) = dmu_array_row_main_cost fi i dmus' (j + 1)  (* c2: cost of recursive call *)
     c3 = 2 · n + 2                                    (* c3: local costs of function *)
   in (res, c1 + c2 + c3))                             (* sum up costs *)
```

```
partial_function (tailrec) basis_reduction_main_cost where
  basis_reduction_main_cost state c = (if i < m
   then let (state', c1) = basis_reduction_step_cost state
     in basis_reduction_main_cost state' (c + c1)
   else (state, c))
```

The function *dmu_array_row_main_cost* is the usual case: one part invokes sub-algorithms or makes a recursive call and extracts the cost by pattern matching on pairs (*c1* and *c2*), one does some local operations and manually annotates the costs for them (*c3*), and, finally, the pair of the computed result and the total cost is returned.

The function *basis_reduction_main_cost* is a bit more interesting. All other functions can easily be annotated without changing their input arguments. *basis_reduction_main* was defined as a *tail-recursive* **partial_function** (see [5]). In order to write *basis_reduction_main_cost* as tail-recursive we add an accumulator *c* to its input arguments. The proof that *basis_reduction_main_cost* is terminating is similar to the termination proof of *basis_reduction_main*. Both functions only terminate if certain preconditions are met (*LLL_invariant*). The proofs on both functions also use *LLL_measure* which measures the number of loop iterations and is bound by a polynomial expression in *m*, *n* and the squared norm of the largest vector.

For both of these cost functions (and all other cost functions) we prove that *result* returns the same value as the corresponding function, and give upper bounds for the return values of *cost*. We end up with the Lemma *reduce_basis_cost_expanded* mentioned in the introduction.

5 Bounds on the Numbers in the LLL Algorithm

Whereas the previous section provides a formally verified upper bound on the number of arithmetic operations, in this section we consider the costs of each individual arithmetic operation, and formally derive bounds on the f_i, $\tilde{\mu}_{i,j}$, and $\tilde{g}_i$, as well as on the auxiliary values computed by Algorithms 2 and 3. Although the implementation of Algorithm 2 computes neither g_i nor $\tilde{g}_i$ throughout its execution, the proof of an upper bound on $\tilde{\mu}_{i,j}$ uses an upper bound on g_i.

Whereas the bounds for g_i will be valid throughout the whole execution of the algorithm, the bounds for the f_i depend on whether we are inside or outside the for-loop in lines 3–4 of Algorithm 1. Within the for-loop, the value of the $\|f_i\|$ can get slightly larger than outside the loop.

To formally verify bounds on the numbers, we first define a stronger LLL-invariant which includes the conditions *f_bound outside fs* and *g_bound gs*. Recall that *A* is the maximum squared norm of the initial f vectors.

definition *f_bound outside k fs* = ($\forall$ *i* < *m*. $\|$*fs* ! *i*$\|^2 \leq$
(**if** *outside* $\vee$ *k* $\neq$ *i* **then** *A* $\cdot$ *m* **else** $4^{m-1} \cdot A^m \cdot m^2$))
definition *g_bound gs* = ($\forall$ *i* < *m*. $\|$*gs* ! *i*$\|^2 \leq$ *A*)
definition *LLL_bound_invariant outside* (*i*, *fs*, *gs*) =
(*LLL_invariant i fs* $\wedge$ *f_bound outside i fs* $\wedge$ *g_bound gs*)

Note that *LLL_bound_invariant* does not enforce a bound on the $\tilde{\mu}_{i,j}$, since such a bound can be derived from the bounds on f, g, and the Gramian determinants.

lemma *mu_bound_Gramian_determinant:*
assumes *j* < *i* **and** *i* < *m*
shows $(\mu\ i\ j)^2 \leq$ *Gramian_determinant fs j* $\cdot \|$*fs* ! *i*$\|^2$

The proof of this fact is rather straightforward and follows closely the one from [25, Chapter 16]. The proof uses Cauchy's inequality ($\|u \bullet v\|^2 \leq \|u\|^2 \cdot \|v\|^2$), which is part of our vector library (the Isabelle theory file `Norms.thy`).

Bounds on the Gramian determinants can be directly derived from the Lemma *Gramian_determinant* and *g_bound gs*:

lemma *Gramian_determinant_bound:*
assumes *LLL_invariant* (*i*, *fs*, *gs*) **and** *g_bound gs* **and** *k* < *m*
shows *Gramian_determinant fs k* $\leq A^k$

The above two lemmas clearly give an upper-bound in terms of *A* on $\tilde{\mu}_{i,j} = d_{j+1}\mu_{i,j} =$ *Gramian_determinant fs* (*Suc j*) $\cdot\ \mu$ *i j*. A bound on the $\tilde{g}$ vectors is obtained similarly from the last lemma and the invariant *g_bound gs*. Bounds in terms of *A* on the intermediate values of σ and τ in Algorithms 2 and 3 are obtained in a straight-forward manner.

Finally, we note that the polynomial complexity and number bounds for Algorithm 1 were not formalized in [5], but that such bounds do follow along the same lines as the ones shown in this and the previous section, with one exception: When g_i is a vector of rationals, an invariant bound on the size of its absolute value does not imply a bound on the numerators and denominators of its components. A similar comment applies to the rational $\mu_{i,j}$ values. To obtain these bounds on numerators and denominators, we use the fact that multiplying g_i (or $\mu_{i,j}$) by the corresponding Gramian determinant, results in an integer-valued vector (or an integer, respectively). This immediately implies bounds on the denominators of the vector components, which, together with the bound on the norm of the vector (given by *g_bound gs*), then implies a bound on the numerators as well. These proofs also require Cramer's lemma (see Section 6).

6 Applying Cramer's Lemma in Isabelle

Cramer's lemma (also known as Cramer's rule) states that, given a system of linear equations $Mx = b$, where M is a non-singular $(n \times n)$-matrix, the unique solution of the system is given by $x_j = \frac{\det M_j}{\det M}$, where M_j is the matrix obtained from M by replacing the j-th column with the vector b. Using Cramer's lemma to solve a system of linear equations is, mathematically, a simple matter, which is why the details of applying it are hand-waved in the informal proofs in [7]. Of course, in the context of proof formalization, all of these details need to be specified. As it turns out, there are also some Isabelle-specific technical issues with using this lemma. In this section, we look at how these issues can be overcome, and also show how the missing details of several proofs described in the previous sections were added in our Isabelle implementation.

Although Cramer's lemma is already available in the Isabelle distribution, there is a technical obstacle to deal with before we can use it in our proof.

lemma *cramer_lemma:* **fixes** *A :: 'a^'n^'n*
 shows *det* (*replace_col_hma A* (*A* $\cdot_v$ *x*) *j*) = *x $ j* $\cdot$ *det A*

The problem is that the lemma is available in HOL-analysis, which uses Harrison's technique to represent vector dimensions via type variables [8]. In contrast, the whole LLL algorithm is formalized using a matrix- and vector-library of the AFP [21]. Here, the dimensions of a matrix *M* of e.g. type *'a mat* are not fixed in the type. Instead we have to add the assumption *M* ∈ *carrier_mat n n* if *M* is of dimensions *n* × *n*.

A recent development in Isabelle/HOL allows us to transfer theorems between the two matrix libraries [3, Section 4]. It is based on local type definitions [10] and Isabelle's transfer mechanism [9]. In order to move a lemma from one matrix representation to the other, *transfer rules* have to be developed for all constants within that lemma. For instance, for Cramer's lemma we must establish the following transfer rule between the constants *replace_col* and *replace_col_hma*. Here, *replace_col A v i* is the matrix *A* where column *i* is replaced by vector *v* using the AFP matrix representation, and *replace_col_hma* provides the same functionality using the HOL-analysis matrix library.

lemma *HMA_M_replace_col [transfer_rule]:*
 (*HMA_M* ===> *HMA_V* ===> *HMA_I* ===> *HMA_M*)
  replace_col replace_col_hma

This transfer rule states that if all three arguments of *replace_col* and *replace_col_hma* are related, then also the result is related. To be more precise, the first arguments of both functions must represent the same matrix (related by *HMA_M*), the second arguments must represent the same vector (related by *HMA_V*), and the third arguments must represent the same index (related by *HMA_I*), in order to conclude that both results represent the same matrix (related by *HMA_M*).

The transfer rule is easy to prove, and afterwards transfer rules for all constants in Cramer's lemma are available, since the existing library [3] already contains transfer rules for determinants, matrix-vector-multiplication, etc. At this point, Cramer's lemma can be transferred immediately. The following Isabelle source code contains the full proof for lemma *cramer_lemma_mat*.

lemma *cramer_lemma_mat:* **fixes** *A :: 'a mat*
 assumes *A* ∈ *carrier_mat n n* **and** *x* ∈ *carrier_vec n* **and** *j* < *n*
 shows *det* (*replace_col A* (*A* $\cdot_v$ *x*) *j*) = *x $ j* $\cdot$ *det A*

using *assms cramer_lemma[untransferred, cancel_card_constraint]*
by *auto*

Here, the *untransferred* attribute of the transfer package [9] transforms *cramer_lemma* into a statement which uses AFP matrices. But since this statement will contain the expression *CARD('n)* – the cardinality of the type *'n* – to represent the dimension, we use *cancel_card_constraint* in order to replace *CARD('n)* by a fresh variable *n* in *cramer_lemma_mat*. This replacement internally relies upon local type definitions [10], since one has to prove that for every natural number *n > 0* a suitable type *'n* exists such that *n = CARD('n)*.

We turn to the missing proof steps involving Cramer's lemma, and how they were formalized in Isabelle. In the previous sections we needed formal proofs of statements like $d_{i+1}g_i \in \mathbb{Z}^n$ and $d_{j+1}\mu_{i,j} \in \mathbb{Z}$, in order to show that certain algorithms only need to store integers, as well as to prove bounds on the sizes of numbers being computed throughout the execution of Algorithm 1.

In the case of $d_{i+1}g_i \in \mathbb{Z}^n$, we first prove that g_i can be written as a sum involving only the f vectors, namely, that $g_i = f_i - \sum_{j<i} \lambda_{i,j} f_j$. Although the existence of such values $\lambda_{i,j}$ is mathematically trivial,[2] in Isabelle we construct these $\lambda_{i,j}$ via a somewhat tedious inductive process. Now, since the f vectors are integer-valued, it suffices to show that $d_{i+1}\lambda_{i,j} \in \mathbb{Z}$, in order to get that $d_{i+1}g_i \in \mathbb{Z}^n$. To prove this, observe that each g_i is orthogonal to every f_l with $l < i$ and therefore $0 = f_l \bullet g_i = f_l \bullet f_i - \sum_{j<i} \lambda_{i,j}(f_l \bullet f_j)$. So the $\lambda_{i,j}$ form a solution to a system of linear equations:

$$\underbrace{\begin{pmatrix} f_1 \bullet f_1 & \dots & f_1 \bullet f_{i-1} \\ \vdots & \ddots & \vdots \\ f_{i-1} \bullet f_1 & \dots & f_{i-1} \bullet f_{i-1} \end{pmatrix}}_{=M=\textit{Gramian_matrix fs i}} \cdot \underbrace{\begin{pmatrix} \lambda_{i,1} \\ \vdots \\ \lambda_{i,i-1} \end{pmatrix}}_{=L} = \underbrace{\begin{pmatrix} f_1 \bullet f_i \\ \vdots \\ f_{i-1} \bullet f_i \end{pmatrix}}_{=F}$$

The coefficient matrix M on the left-hand side where $M_{i,j} = f_i \bullet f_j$ is exactly the Gramian matrix of *fs* and *i*. By an application of Cramer's lemma, we deduce:

$$\begin{aligned} \lambda_{i,j} \cdot \textit{det (Gramian_matrix fs i)} &= L \ \$ \ j \cdot \textit{det } M \\ &= \textit{det (replace_col } M \ (M \cdot_v L) \ j) \\ &= \textit{det (replace_col } M \ F \ j) \end{aligned}$$

The matrix *replace_col* M F *j* contains only inner products of the f vectors as entries and these are of course integers. Then the determinant is also an integer and $\lambda_{i,j} \cdot$ *Gramian_determinant fs i* $\in \mathbb{Z}$. Unfolding the definition of g_i, where $g_i = f_i - \sum_{j<i} \lambda_{i,j} f_j$ in *Gramian_determinant fs i* $\cdot_v$ g_i, leaves us with sums and differences consisting of only integers.

Since $\mu_{i,j} = \frac{f_i \bullet g_j}{\|g_j\|^2}$ and $\frac{d_{j+1}}{d_j} = \|g_j\|^2$, the statement $d_{j+1}\mu_{i,j} \in \mathbb{Z}$ is easily deduced from $d_{i+1}g_i \in \mathbb{Z}^n$, without a separate application of Cramer's lemma. The lemma is used again when proving that the σ values in Algorithm 2 (and the τ values in Algorithm 3) are integers. In the case of the σ values, we show that $d_{l+1}(f_i - \sum_{j<l} \mu_{i,j} g_j)$ is integer-valued (note that the sum only goes up to l, not i), a case that is shown similarly as $d_{i+1}g_i \in \mathbb{Z}^n$, except with Cramer's lemma applied to an l-dimensional matrix rather than an i-dimensional one.

[2] Since $g_i = f_i - \sum_{j<i} \mu_{i,j} g_j$, and, by the construction of the g's, $g_0, \dots, g_{i-1}$ and $f_0, \dots, f_{i-1}$ span the same space, the $\lambda_{i,j}$ are simply the coordinates of $\sum_{j<i} \mu_{i,j} g_j$ in the (not necessarily orthogonal) basis formed by the first i of the f vectors.

7 Conclusion

We have extended the original formalization of the LLL basis reduction algorithm from [5], by also formalizing a more efficient version of the algorithm, and by giving formal proofs of the polynomial-time complexity of both the new implementation and of the old one. As the performance of the new implementation is comparable to that of some commercial products that implement the same algorithm, this puts our verified implementation within the realm of practically usable software. One way to further build on this work would be to formalize a fast polynomial factorization algorithm that uses the LLL basis reduction algorithm as a subroutine, such as van Hoeij's algorithm [24], which would make full use of the efficiency of our current implementation.

Acknowledgments

We thank the anonymous reviewers for their helpful feedback. This research was supported by the Austrian Science Fund (FWF) project Y757. The authors are listed in alphabetical order regardless of individual contributions or seniority.

References

[1] R. Bottesch, J. Divasón, M. Haslbeck, S. Joosten, R. Thiemann, and A. Yamada. A verified LLL algorithm. *Archive of Formal Proofs*, Feb. 2018. http://isa-afp.org/entries/LLL_Basis_Reduction.html, Formal proof development.

[2] C. Cohen. Construction of real algebraic numbers in Coq. In *ITP 2012*, volume 7406 of *LNCS*, pages 67–82, 2012.

[3] J. Divasón, S. Joosten, O. Kunčar, R. Thiemann, and A. Yamada. Efficient certification of complexity proofs: Formalizing the Perron–Frobenius theorem (invited talk paper). In *CPP 2018*, pages 2–13. ACM, 2018.

[4] J. Divasón, S. Joosten, R. Thiemann, and A. Yamada. A formalization of the Berlekamp–Zassenhaus factorization algorithm. In *CPP 2017*, pages 17–29. ACM, 2017.

[5] J. Divasón, S. Joosten, R. Thiemann, and A. Yamada. A formalization of the LLL basis reduction algorithm. In *ITP 2018*, volume 10895 of *LNCS*, pages 160–177, 2018.

[6] M. Eberl, M. W. Haslbeck, and T. Nipkow. Verified analysis of random binary tree structures. In *ITP 2018*, volume 10895 of *LNCS*, pages 196–214, 2018.

[7] U. Erlingsson, E. Kaltofen, and D. Musser. Generic Gram–Schmidt orthogonalization by exact division. In *ISSAC 1996*, pages 275–282. ACM, 1996.

[8] J. Harrison. The HOL light theory of Euclidean space. *J. Autom. Reasoning*, 50(2):173–190, 2013.

[9] B. Huffman and O. Kunčar. Lifting and transfer: A modular design for quotients in Isabelle/HOL. In *CPP 2013*, volume 8307 of *LNCS*, pages 131–146, 2013.

[10] O. Kunčar and A. Popescu. From types to sets by local type definitions in higher-order logic. In *ITP 2016*, volume 9807 of *LNCS*, pages 200–218, 2016.

[11] A. K. Lenstra, H. W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:515–534, 1982.

[12] W. Li and L. C. Paulson. A modular, efficient formalisation of real algebraic numbers. In *CPP 2016*, pages 66–75. ACM, 2016.

[13] D. Micciancio. The shortest vector in a lattice is hard to approximate to within some constant. *SIAM J. Comput.*, 30(6):2008–2035, 2000.

[14] P. Q. Nguyen and B. Vallée, editors. *The LLL Algorithm – Survey and Applications*. Information Security and Cryptography. Springer, 2010.

[15] T. Nipkow. Verified root-balanced trees. In *APLAS 2017*, volume 10695 of *LNCS*, pages 255–272, 2017.

[16] T. Nipkow and G. Klein. *Concrete Semantics*. Springer, 2014.

[17] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[18] A. Storjohann. Faster algorithms for integer lattice basis reduction. *Technical report/Department of Computer Science, ETH Zurich*, 249, 1996.

[19] The FPLLL development team. fplll, a lattice reduction library. Available at https://github.com/fplll/fplll, 2016.

[20] R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *TPHOLs'09*, volume 5674 of *LNCS*, pages 452–468, 2009.

[21] R. Thiemann and A. Yamada. Matrices, Jordan normal forms, and spectral radius theory. *Archive of Formal Proofs*, Aug. 2015. http://isa-afp.org/entries/Jordan_Normal_Form.html, Formal proof development.

[22] R. Thiemann and A. Yamada. Algebraic numbers in Isabelle/HOL. In *ITP 2016*, volume 9807 of *LNCS*, pages 391–408, 2016.

[23] R. Thiemann and A. Yamada. Formalizing Jordan normal forms in Isabelle/HOL. In *CPP 2016*, pages 88–99. ACM, 2016.

[24] M. van Hoeij. Factoring polynomials and the knapsack problem. *J. Number Theory*, 95:167–189, 2002.

[25] J. von zur Gathen and J. Gerhard. *Modern computer algebra (3rd ed.)*. Cambridge University Press, 2013.