Generalization of the Integration of Symbols Algorithm for non rational alphabets

Maximilian Danner

29. Oktober 2024

1 The symbol alphabet

1.1 Introduction

(i) We are given a symbol alphabet

$$\mathscr{L} = \{l_1, ..., l_n\} \tag{1}$$

which consists of expressions with rational functions and possibly square roots of rational functions as well as rational constants. We assume $\mathscr{L}$ to be multiplicatively independent; i.e. there shall be no factorization of any l_i in terms of elements from $\mathscr{L} - \{l_i\}$. Our goal is to construct admissible arguments f_i for the Li-functions. Here, admissibility means the same thing as in the rational case: for depth 1 arguments, $1 - f_i$ has to factorize over $\mathscr{L}$, for depth 2 arguments $1 - f_i f_j$ has to factorize etc. (see thesis). Factorizability is understood in the following sense:

$$\exists\, r_i \in \mathbb{Q} : \; 1 - f_i = \pm \prod_{i=1}^{n} l_i^{r_i} \tag{2}$$

with $r_i \in \mathbb{Q}$ a priori. The arguments are to be constructed from the set $\mathscr{L}$ by testing products

$$f = \prod_{i=1}^{n} l_i^{a_i}, \quad a_i \in \mathbb{Q} \tag{3}$$

for admissibility. This is in general a very hard task for two reasons:

- There are very many possible products to check.
- The factorization of $1 - f$ over $\mathscr{L}$ is hard to check.

(ii) A heuristic factorization algorithm. Suppose you have an expression g that should be factorized over $\mathscr{L}$ in a way similar to equation 3. To do this, take the logarithm of this equation

$$\log(g) - \sum_{i=1}^{n} a_i \log(l_i) = 0. \tag{4}$$

Find an instance of the variables such that $l_1, ..., l_n$ evaluate to different numbers (practically, a random point will usually do the job). Then, employ an integer relation algorithm (PSLQ, LLL, ...) to find the coefficients $a_1, ..., a_n$.

It is hard to beat this algorithm:

- It is based only on numerical as opposed to symbolic computations. All expressions can easily and very efficiently be computed to arbitrary precision; and lattice algorithms are well developed.
- Factorization theory of generalized polynomials is developed for expressions like $\sum_{i=1}^{n} a_i x^{r_i}$ (and there only from the abstract algebra perspective) - but not for expressions like $\sum_{i=1}^{n} a_i (x + b_i)^{r_i}$. Multivariate expressions of this kind are not discussed at all.

– Our factorization task is somewhat special: we don't want to find some factorization in terms of *arbitrary* factors but rather a factorization *over the set*

$$\{l^r \mid l \in \mathscr{L},\ r \in \mathbb{Q}\}, \tag{5}$$

where, typically, $\deg(r) = |n| + |m|$ with $r = n/m,\ n, m \in \mathbb{Z},\ \gcd(n, m) = 1$, is rather small. A heuristic algorithm seems to be a better choice compared to a full-fledged algorithm based on theoretical considerations, if one exists.

Note: we do not ask for uniqueness of the factorization! It suffices to know whether there exists one or not.

1.2 Preparational step: construction of a suitable alphabet

Note: The following is only relevant if the alphabet contains letters with square-roots.

We will discuss some of the steps described in https://arxiv.org/pdf/1907.00491.pdf. Our goal for this section is a „basis change" (we'll discuss the quotation marks in a minute) on the vector space

$$\mathrm{span}_{\mathrm{mult}}(\mathscr{L}) := \left\{\prod_{i=1}^{n} l_i^{r_i} \mid l_i \in \mathscr{L} \ \wedge\ r_i \in \mathbb{Q}\right\}, \tag{6}$$

$$\left(\prod_{i=1}^{n} l_i^{r_i}\right) \oplus \left(\prod_{j=1}^{n} l_j^{s_j}\right) := \prod_{i=1}^{n} l_i^{r_i+s_i}, \quad \lambda \odot \left(\prod_{i=1}^{n} l_i^{r_i}\right) := \prod_{i=1}^{n} l_i^{\lambda r_i}, \tag{7}$$

such that it suffices in equation 3 to consider only $a_i \in \mathbb{Z}$. The above mentioned paper describes roughly the following steps which I present in the more generalized way that I implemented in C++.

(i) Write $\mathscr{L}$ as a disjoint union $\mathscr{L}_R \cup \mathscr{L}_A$ where $\mathscr{L}_R$ denotes the rational letters and $\mathscr{L}_A$ the non-rational, algebraic ones. Let $r_1, ..., r_w$ be the square roots occurring in $\mathscr{L}_A$. Suppose, $\mathscr{L}_A$ is such that the following two conditions are met:

– For all $l \in \mathscr{L}_A$ the conjugate letter $\bar{l}$ (obtained by letting $r \to -r$) factorizes over $\mathscr{L}$.

– For all $l \in \mathscr{L}_A$ the product $l\bar{l}$ factorizes over $\mathscr{L}_R$.

In this case we can proceed with the following steps and construct an improved version of $\mathscr{L}_A$, denoted by $\tilde{\mathscr{L}}_A$.

(ii) Generate the set of products of square roots up to the number of factors n_r:

$$\mathscr{R} := \left\{\prod_{i=1}^{w} r_i^{a_i} \mid a_i \in \mathbb{N}_0,\ \sum_i a_i \le n_r\right\} \tag{8}$$

Similarly, define

$$\mathscr{M}_d := \left\{\pm \prod_{i=1}^{|\mathscr{L}_R|} l_i^{a_i} \mid l_i \in \mathscr{L}_R,\ a_i \in \mathbb{N}_0,\ \sum_i a_i \le n_d\right\} \tag{9}$$

and

$$\mathscr{M}_n := \left\{\pm \prod_{i=1}^{|\mathscr{L}_R|} l_i^{a_i} \mid l_i \in \mathscr{L}_R,\ a_i \in \mathbb{N}_0,\ \sum_i a_i \le n_n\right\}. \tag{10}$$

The parameters n_r, n_d, n_n (r for roots, d for denominator and n for numerator) are to be specified by the user and are usually ≤ 10.

(iii) The main algorithm works as sketched in algorithm 1. One important difference is, however, that all calculations are carried out not symbolically but numerically (after evaluating the appropriate symbolic expressions at some random point). This is for performance reasons. The outer for-loop is parallelized using process-parallelism (since GiNaC is not thread-safe and hence multithreading is not an option). The main idea of the algorithm is to make an ansatz $l = q + r$ where q is some rational function and $r \in \mathscr{R}$ such that $l\bar{l}$ factorizes over $\mathscr{L}_R$; i.e. $(q + r)(q - r) = q^2 - r^2 = \frac{f_n}{f_d}, f_n \in \mathscr{M}_n, f_d \in \mathscr{M}_d$. The algorithm kind of reverses the logic: it

computes $\frac{f_n}{f_d} + r^2$ and checks that this is a perfect square q^2 (numerically, using the `cln::sqrtp()` function from the CLN library). Two remarks:

- The result of this algorithm is in general not multiplicatively independent. One has to further reduce it using, e.g. the PSLQ-algorithm.
- One should furthermore check that every element in $\mathscr{L}_A$ can be written as a product of integer powers of elements in $\tilde{\mathscr{L}}_A$ and vice versa - again, e.g., via the PSLQ algorithm. If this is possible this also justifies calling the procedure a basis change.

We denote the new improved alphabet by $\tilde{\mathscr{L}} := \mathscr{L}_R \cup \tilde{\mathscr{L}}_A$ - a good sanity check is $|\mathscr{L}| = |\tilde{\mathscr{L}}|$.

Algorithm 1: Main algorithm

Data: Input data $\mathscr{R}$, $\mathscr{M}_d$, $\mathscr{M}_n$.
Result: Candidates for $\tilde{\mathscr{L}}_A$ stored in list result.

```
for r ∈ R do
    for m_d ∈ M_d do
        f_d ← m_d / denom(r^2)
        if f_d is perfect square then
            for m_n ∈ M_n do
                f_n ← (m_n + num(r^2) f_d) / denom(r^2)
                if f_n is perfect square then
                    cout ≪ "found one!" ≪ endl
                    result.push_back(sqrt(f_n/f_d) + r)
```

2 Construction of admissible arguments

2.1 Depth 1 arguments

Our goal in this section is to find admissible arguments for the Li_n functions as described in the introduction. Since this requires many factorizability checks, we need a strong necessary condition that can be checked efficiently on computers to rule out as many possible arguments as possible from the start, so that the relatively slow (numerical) factorizability check via PSLQ or LLL is performed as rarely as possible.

2.1.1 Background information on the Smith-Decomposition

Definition 1 *Smith normal form.*

- *Let $A = (a_{ij}) \in \mathbb{Z}^{n\times m}$. A is said to be in Smith normal form if $a_{ij} = 0$ for $i \neq j$ and if for $d_i := a_{ii}$ with $r := \min(n, m)$ we have*
$$d_i \geq 0 \text{ (for } i \in \{1, ..., r\}) \quad \text{and} \quad d_i | d_{i+1} \text{ (for } i \in \{1, ..., r-1\}). \tag{11}$$
- *$D \in \mathbb{Z}^{n\times m}$ is called Smith normal form of A if there are $U \in \mathrm{GL}_n(\mathbb{Z})$ and $V \in \mathrm{GL}_m(\mathbb{Z})$ such that $UDV = A$.*

Theorem 1 *Existence and Uniqueness of Smith decomposition.*

- *For all $A \in \mathbb{Z}^{n\times m}$ there is a Smith decomposition $UDV = A$. It can be calculated with an algorithm that terminates after finitely many steps.*
- *D (but not U and V) is uniquely determined by A.*

2.1.2 Background information on the LLL-Algorithm

Definition 2 *Lattices. Let V be a $\mathbb{R}$-vectorspace and $B = \{v_1, ..., v_n\}$ be a basis for V. Then the ($\mathbb{Z}$-)lattice Λ in V generated by B is given by*

$$\Lambda = \left\{ \sum_{i=1}^{n} a_i v_i \mid a_i \in \mathbb{Z} \right\}. \tag{12}$$

The same lattice can be generated by several different bases.

Definition 3 *LLL-reduced bases. A basis $B = \{v_1, ..., v_n\}$ (with Gram-Schmidt orthogonalized $B^* = \{v_1^*, ..., v_n^*\}$) is LLL-reduced if there exists $\delta \in (0.25, 1]$ such that*

- *For all $1 \leq j < i \leq n$:*

$$\left| \frac{\langle v_i, v_j^* \rangle}{\langle v_j^*, v_j^* \rangle} \right| \leq \frac{1}{2}, \tag{13}$$

- *and for $k \in \{2, 3, ..., n\}$:*

$$\delta \|v_{k-1}^*\|^2 \leq \|v_k^*\|^2 + \left(\frac{\langle v_i, v_j^* \rangle}{\langle v_j^*, v_j^* \rangle} \right)^2 \|v_{k-1}^*\|^2. \tag{14}$$

Theorem 2 *LLL-algorithm. There exists a polynomial-time algorithm that reduces a given B to a LLL-reduced basis.*

The LLL-algorithm can be used to find integer relations; i.e. given $x_1, ..., x_n \in \mathbb{R}$, find $a_1, ..., a_n \in \mathbb{Z}$ (with as small absolute values as possible) such that $\sum_i a_i x_i = 0$. This works as follows: First, define a very large number λ (e.g. $\lambda = 10^{80}$). Form $x_i' = \lambda x_i$ and then the matrix

$$B = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & -x_1' \\ 0 & 1 & 0 & \cdots & 0 & -x_2' \\ 0 & 0 & 1 & \cdots & 0 & -x_3' \\ & & & \ddots & & \\ 0 & 0 & 0 & \cdots & 1 & -x_n' \end{pmatrix} \equiv \begin{pmatrix} - & v_1 & - \\ - & v_2 & - \\ - & v_3 & - \\ & \vdots & \\ - & v_n & - \end{pmatrix} \in \mathbb{R}^{n \times (n+1)} \tag{15}$$

Then use the LLL-algorithm on B to get the corresponding LLL-reduced basis. The first n entries of the first vector in this (ordered) LLL-reduced basis are the sought after integers $a_1, ..., a_n$.

2.1.3 Discussion of the algorithm

The algorithm is divided in three parts:

(i) Precomputations for computationally expensive calculations (i.e. Smith decomposition and high-precision evaluation of (the logs) of the letters in the alphabet).

(ii) Preselection of admissible arguments via the Smith decomposition (and reality conditions).

(iii) Final selection of admissible arguments via the LLL-algorithm (Note: the fact that I use the LLL- rather than the PSLQ-algorithm is based on convenience: there are not many C++ - libraries implementing the PSLQ-algorithm in an optimized way; but for the LLL-algorithm there is the well-known and well-tested fpLLL-library. I tried to implement the PSLQ-algorithm myself but this turned out to be too slow).

The whole Smith decomposition process works as follows:

- Say, the letters $l_1, ..., l_n$ depend on variables $x_1, ..., x_d$. First, we have to find rational instances of $x_1, ..., x_d$ (denoted by $x_1^{(i)}, ..., x_d^{(i)}$, $i \in \{1, 2, 3\}$) such that $r_1, ..., r_w$ evaluate to rational numbers (and hence the whole alphabet evaluates to rational numbers). For now, I have (apart from a brute force method that needs to be specialized by hand for the concrete use-case) no systematic way of doing this; in fact, it might not even be always possible. This leads to vectors of rational numbers $\ell_i = (l_1^{(i)}, ..., l_n^{(i)}) := (l_1(x_1^{(i)}, ..., x_d^{(i)}), ..., l_n(x_1^{(i)}, ..., x_d^{(i)}))^T \in \mathbb{Q}^n$.

- We are given an expression $1 - \prod_{j=1}^{n} (l_j^{(i)})^{a_j}$ and ask if there are $b_j \in \mathbb{Z}$ such that

$$1 - \prod_{j=1}^{n} (l_j^{(i)})^{a_j} = \pm \prod_{j=1}^{n} (l_j^{(i)})^{b_j} \iff \left| 1 - \prod_{j=1}^{n} (l_j^{(i)})^{a_j} \right| = \prod_{j=1}^{n} |l_j^{(i)}|^{b_j} \tag{16}$$

It is our goal to render this an integer factorization problem. Since b_i could also be negative we extend $\bar{\ell}_i := (l_1^{(i)}, ..., l_n^{(i)}, 1/l_1^{(i)}, ..., 1/l_n^{(i)})^T \in \mathbb{Q}^{2n}$. For each i, determine the smallest natural number A_i such that $A_i \bar{\ell}_i =: (\tilde{l}_1^{(i)}, ..., \tilde{l}_{2n}^{(i)})^T \in \mathbb{Z}^n$ (obviously, A_i is just the least common multiple of the denominators of the rational numbers in $\bar{\ell}_i$). It is clear that there are always $\tilde{b}_j \in \mathbb{N}_0$, $j \in \{1, ..., 2n\}$ such that

$$\prod_{j=1}^{n} |l_j^{(i)}|^{b_j} = \prod_{j=1}^{2n} |l_j^{(i)}|^{\tilde{b}_j} \cdot \left| \frac{1}{l_j^{(i)}} \right|^{\tilde{b}_{j+n}}. \tag{17}$$

Define $\tilde{b} := \sum_{j=1}^{2n} \tilde{b}_j$ and observe:

$$\left| 1 - \prod_{j=1}^{n} (l_j^{(i)})^{a_j} \right| = \prod_{j=1}^{n} |l_j^{(i)}|^{b_j} \iff A_i^{\tilde{b}} \left| 1 - \prod_{j=1}^{n} (l_j^{(i)})^{a_j} \right| = \prod_{j=1}^{2n} |\tilde{l}_j^{(i)}|^{\tilde{b}_j}. \tag{18}$$

In practice, $\tilde{b}$ is not known. But we can set two bounds:

- $\tilde{b} \geq \tilde{b}_{\min}$ where $\tilde{b}_{\min} \geq 0$ is the smallest natural number such that the left hand side is an integer.
- $\tilde{b} \leq \tilde{b}_{\max}$ where $\tilde{b}_{\max}$ is an empirical parameter to be defined by the user. Typically, $\tilde{b}_{\max} < n$ (in my implementation even $< n/3$).

One has to iterate over $\tilde{b}$ while it is inside the above specified bounds. If there is no factorization found for any of the $\tilde{b}$'s we discard the candidate. Now, with $\tilde{b}_{\min} \leq \tilde{b} \leq \tilde{b}_{\max}$ the right hand side in 18 constitutes a problem of integer factorization where the left hand side is given and we look for the $\tilde{b}_i \in \mathbb{N}_0$.

- We translate this into a problem of linear algebra. To do this, we compute prime factors as follows:

$$A_i^{\tilde{b}} \left| 1 - \prod_{j=1}^{n} (l_j^{(i)})^{a_j} \right| = \prod_{m=1}^{F_i} (p_m^{(i)})^{k_m^{(i)}}, \tag{19}$$

$$\tilde{l}_j^{(i)} = \prod_{m=1}^{F_{i,j}} (p_{j,m}^{(i)})^{t_{j,m}^{(i)}} \tag{20}$$

with $F_i, F_{i,j}, k_m, t_{j,m} \in \mathbb{N}_0$. Since the numbers on the left hand side are rather large (often several dozen digits long) one may worry that their prime factorization may take too long. But by construction of A_i as a least common multiple the prime factors are always rather small (≤ 100000 in all cases I've tested until now) and prime factorization can be done fast with a precomputed list of primes. Clearly, we need

$$\{p_m^{(i)}\}_m \subseteq \{p_{j,m}^{(i)}\}_{j,m} \text{ for all } i \in \{1, 2, 3\} \tag{21}$$

- otherwise factorization is not possible. Construct the union $\mathscr{P}_i := \bigcup_{j,m} \{p_{j,m}^{(i)}\} =: \{s_m^{(i)}\}_m$. Then we can write

$$A_i^{\tilde{b}} \left| 1 - \prod_{j=1}^{n} (l_j^{(i)})^{a_j} \right| = \prod_{m=1}^{|\mathscr{P}_i|} (s_m^{(i)})^{\tilde{k}_m^{(i)}}, \tag{22}$$

$$\tilde{l}_j^{(i)} = \prod_{m=1}^{|\mathscr{P}_i|} (s_m^{(i)})^{\tilde{t}_{j,m}^{(i)}} \tag{23}$$

where the tilde in $\tilde{k}_m^{(i)}$ and $\tilde{t}_{j,m}^{(i)}$ indicates that one probably has to add some zeroes as compared with the non-tilde versions (because of the union). With this and using the uniqueness of prime factorization, equation

18 with 22 and 23 plugged in translates to the following linear systems for $i \in \{1,2,3\}$:

$$\begin{aligned} \tilde{k}_1^{(i)} &= \tilde{b}_1 \tilde{t}_{1,1}^{(i)} + \tilde{b}_2 \tilde{t}_{2,1}^{(i)} + \cdots + \tilde{b}_{2n} \tilde{t}_{2n,1}^{(i)} \\ \tilde{k}_2^{(i)} &= \tilde{b}_1 \tilde{t}_{1,2}^{(i)} + \tilde{b}_2 \tilde{t}_{2,2}^{(i)} + \cdots + \tilde{b}_{2n} \tilde{t}_{2n,2}^{(i)} \\ &\vdots \\ k_{|\mathscr{P}_i|}^{(i)} &= \tilde{b}_1 \tilde{t}_{1,|\mathscr{P}_i|}^{(i)} + \tilde{b}_2 \tilde{t}_{2,|\mathscr{P}_i|}^{(i)} + \cdots + \tilde{b}_{2n} \tilde{t}_{2n,|\mathscr{P}_i|}^{(i)} \end{aligned} \tag{24}$$

With the obvious identifications this results in the three matrix equations $k_i = T_i b$ with $k_i \in \mathbb{Z}^{r_i}$, $T_i \in \mathbb{Z}^{r_i \times c_i}$, $b \in \mathbb{Z}^{c_i}$ where $r_i := |\mathscr{P}_i|$, $c_1 = c_2 = c_3 = 2n$ (note that b does *not* depend on i).

- We now want to use the fact that b does not depend on i to determine (rather strong) compatibility equations that have to be fulfilled if a factorization exists. The first step is to combine the three matrix equations into just one:

$$\underbrace{\begin{pmatrix} k_1 \\ k_2 \\ k_3 \end{pmatrix}}_{=:\,\mathscr{k}} = \underbrace{\begin{pmatrix} T_1 & 0 & 0 \\ 0 & T_2 & 0 \\ 0 & 0 & T_3 \end{pmatrix}}_{=:\,\mathscr{T}} \underbrace{\begin{pmatrix} b \\ b \\ b \end{pmatrix}}_{=:\,\mathscr{b}} \tag{25}$$

Now we use the Smith decomposition multiple times:

1. $U_1 D_1 V_1 = \mathscr{T}$. Then the above equation is equivalent to $D_1 x_1 = y_1$ with $x_1 = V_1 \mathscr{b}$ and $y_1 = U_1^{-1} \mathscr{k}$. Dimensions:

$$\mathscr{T}, D_1 : (r_1 + r_2 + r_3) \times 3c_1, \quad U_1 : (r_1 + r_2 + r_3) \times (r_1 + r_2 + r_3), \quad V_1 : 3c_1 \times 3c_1. \tag{26}$$

Write (usually $r_i \le c_i$):

$$D_1 = \left(\begin{array}{ccccccc|ccc} D_1^{(1)} & 0 & \cdots & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & D_1^{(2)} & \cdots & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & & \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & D_1^{(k_1)} & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & & \vdots & \vdots & \ddots & \vdots & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \end{array}\right) \begin{matrix} \left.\vphantom{\begin{matrix} a \\ a \\ a \\ a \\ a \\ a \\ a \end{matrix}}\right\} r_1 + r_2 + r_3 \\ \\ \\ \\ \\ \\ \\ \end{matrix} \quad \text{and} \quad x_1 = \begin{pmatrix} \hat{x}_1^{(1)} \\ \vdots \\ \hat{x}_1^{(k_1)} \\ \lambda_1^{(k_1+1)} \\ \vdots \\ \lambda_1^{(3c_1)} \end{pmatrix} \tag{27}$$

$$\underbrace{\qquad\qquad\qquad}_{r_1 + r_2 + r_3} \qquad \underbrace{\qquad\qquad}_{3c_1 - (r_1 + r_2 + r_3)}$$

Now $\hat{x}_1^{(i)} = y_1^{(i)} / D_1^{(i)}$ for $1 \le i \le k_1$; $\lambda_1^{(i)}$ are some yet to be determined parameters. If $y_1^{(i)} \neq 0$ for $i > k_1$ or $y_1^{(i)} \nmid D_1^{(i)}$ for $i \le k_1$, then there does not exist an integer solution and we can skip the further calculation, $\tilde{b} \leftarrow \tilde{b} + 1$.

As a next step, rewrite $\mathscr{b} = V_1^{-1} x_1$:

$$\begin{pmatrix} b \\ b \\ b \end{pmatrix} = \left(\begin{array}{c|c} V_{11} & V_{12} \\ \hline V_{21} & V_{22} \\ \hline V_{31} & V_{32} \end{array}\right) \begin{pmatrix} \hat{x}_1 \\ \lambda_1 \end{pmatrix} \tag{28}$$

$$\underbrace{\qquad}_{k_1} \quad \underbrace{\qquad\qquad}_{3c_1 - k_1}$$

From this the following equations follow

$$\begin{aligned} &\Delta_1 \lambda_1 = \beta_1, \quad \text{where} \quad \Delta_1 = V_{12} - V_{22} \text{ and } \beta_1 = (V_{21} - V_{11})\hat{x}_1, \\ &\Delta_2 \lambda_2 = \beta_2, \quad \text{where} \quad \Delta_2 = V_{22} - V_{32} \text{ and } \beta_2 = (V_{31} - V_{21})\hat{x}_1. \end{aligned} \tag{29}$$

Note that $\Delta_i \in \mathbb{Z}^{c_1 \times (3c_1 - k_1)}$ and $\beta_i \in \mathbb{Z}^{c_1}$.

2. $U_2 D_2 V_2 = \Delta_1$. The first equation in 29 is then equivalent to $D_2 x_2 = y_2$ with $x_2 = V_2 \lambda_1$ and $y_2 = U_2^{-1} \beta_1$. Dimensions:

$$\Delta_1, D_2 : c_1 \times (3c_1 - k_1), \quad U_2 : c_1 \times c_1, \quad V_2 : (3c_1 - k_1) \times (3c_1 - k_1). \tag{30}$$

Again, write

$$D_2 = \left(\begin{array}{cccccccc|ccc} D_2^{(1)} & 0 & \cdots & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & D_2^{(2)} & \cdots & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & & \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & D_2^{(k_2)} & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & & \vdots & \vdots & \ddots & \vdots & 0 & \cdots & 0 \\ \underbrace{0}_{c_1} & 0 & \cdots & 0 & 0 & \cdots & 0 & \underbrace{0}_{2c_1-k_1} & \cdots & 0 \end{array}\right) \begin{matrix} \\ \\ \\ \\ \\ \\ \} c_1 \end{matrix} \quad \text{and} \quad x_2 = \begin{pmatrix} \hat{x}_2^{(1)} \\ \vdots \\ \hat{x}_2^{(k_2)} \\ \lambda_2^{(k_2+1)} \\ \vdots \\ \lambda_2^{(3c_1-k_1)} \end{pmatrix} \tag{31}$$

Now $\hat{x}_2^{(i)} = y_2^{(i)}/D_2^{(i)}$ for $1 \le i \le k_2$; $\lambda_2^{(i)}$ are some yet to be determined parameters. If $y_2^{(i)} \neq 0$ for $i > k_2$ or $y_2^{(i)} \nmid D_2^{(i)}$ for $i \le k_2$, then there does not exist an integer solution and we can skip the further calculation, $\tilde{b} \leftarrow \tilde{b} + 1$.

Rewrite the second equation in 29 by using this result: $\Delta_2 V_2^{-1} x_2 = \beta_2$. Set $\Omega := \Delta_2 V_2^{-1}$ and introduce the following splitting:

$$\left(\underbrace{\Omega_1}_{k_2} \mid \underbrace{\Omega_2}_{3c_1-k_1-k_2}\right) \begin{pmatrix} \hat{x}_2 \\ \lambda_2 \end{pmatrix} = \beta_2 \iff \Omega_2 \lambda_2 = \beta_2 - \Omega_1 \hat{x}_1 =: \gamma \in \mathbb{Z}^{c_1}. \tag{32}$$

3. $U_3 D_3 V_3 = \Omega_2$. Then equation 32 is equivalent to $D_3 x_3 = y_3$ with $x_3 = V_3 \lambda_2$ and $y_3 = U_3^{-1} \gamma_1$. Dimensions:

$$\Omega_2, D_3 : c_1 \times (3c_1 - k_1 - k_2), \quad U_3 : c_1 \times c_1, \quad V_3 : (3c_1 - k_1 - k_2) \times (3c_1 - k_1 - k_2). \tag{33}$$

Again, write

$$D_3 = \left(\begin{array}{cccccccc|ccc} D_3^{(1)} & 0 & \cdots & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & D_3^{(2)} & \cdots & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & & \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & D_3^{(k_3)} & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & & \vdots & \vdots & \ddots & \vdots & 0 & \cdots & 0 \\ \underbrace{0}_{c_1} & 0 & \cdots & 0 & 0 & \cdots & 0 & \underbrace{0}_{2c_1-k_1-k_2} & \cdots & 0 \end{array}\right) \begin{matrix} \\ \\ \\ \\ \\ \\ \} c_1 \end{matrix} \quad \text{and} \quad x_3 = \begin{pmatrix} \hat{x}_3^{(1)} \\ \vdots \\ \hat{x}_3^{(k_3)} \\ \lambda_3^{(k_3+1)} \\ \vdots \\ \lambda_3^{(3c_1-k_1-k_2)} \end{pmatrix} \tag{34}$$

Now $\hat{x}_3^{(i)} = y_3^{(i)}/D_3^{(i)}$ for $1 \le i \le k_3$; $\lambda_3^{(i)}$ are some yet to be determined parameters. If $y_3^{(i)} \neq 0$ for $i > k_3$ or $y_3^{(i)} \nmid D_3^{(i)}$ for $i \le k_3$, then there does not exist an integer solution and $\tilde{b} \leftarrow \tilde{b} + 1$. Otherwise, the condition we wanted to determine is set to true and we exit the preselection step.

Note that all the computationally expensive stuff (Smith decomposition, matrix inversion, matrix-matrix-product) can be precomputed and this only once because all of it does not depend on k. The only calculations we have to do in the main work loop are matrix-vector-products and logical operations. From running the algorithm we see that the condition from equation 21 together with the ones from 1., 2., 3. is quite strong.

The LLL-part is straightforward as described in the section 3.2.

Some further remarks regarding the implementation:

- We have to loop through all possible (in our case numerical) expressions of the form $1 - \prod_{j=1}^{n} (l_j^{(i)})^{a_j}$. This is rather costly, especially for a larger alphabet. I have optimized this as follows:
  - The main information of what value $1 - \prod_{j=1}^{n} (l_j^{(i)})^{a_j}$ takes lies in the vector of exponents $(a_1, ..., a_n)$. Again, there has to be some kind of user defined cutoff N; in this case the cutoff is defined via the condition $\sum_{i=1}^{n} |a_i| \le N$. I have written a function `sum_abs_values` that (rather efficiently) generates all possible vectors $(a_1, ..., a_n) \in \mathbb{Z}^n$ (called partitions in the implementation) that satisfy this constraint modulo two equivalence relations: two partitions are considered equal (for the purpose of `sum_abs_values`) if one is the negative of the other or one is a permutation of the other.

  - The first equivalence relation is justified since if $1 - \prod_{j=1}^{n} (l_j^{(i)})^{a_j}$ factorizes over $\tilde{\mathscr{L}}$ then so does $1 - \left(\prod_{j=1}^{n} (l_j^{(i)})^{a_j}\right)^{-1}$ - we only need to check one of them.
  - The second equivalence relation is justified since we also loop through all permutations of the partitions. This is done in a parallel manner - again using process parallelism.

- For each concrete instance of the current permutation of the current partition we check $1 - \prod_{j=1}^{n} (l_j^{(i)})^{a_j}$ as well as $1 + \prod_{j=1}^{n} (l_j^{(i)})^{a_j}$ for factorizability.

- If one wishes, one can also impose a reality condition (in this case $\prod_{j=1}^{n} (l_j^{(i)})^{a_j} \leq 1$) for a certain region of the kinematical parameters in order to exclude some candidates right from the beginning. I think this should be done only *after* one has used all results from the construction of depth 1 arguments for the construction of higher depths arguments (but there one should be able to directly impose a reality condition).

2.2 Higher depth arguments

We again want to construct all arguments for $\mathrm{Li}_{n_1,...,n_d}$, $d \geq 2$, functions such that the symbol factors factorize over the given alphabet. It is not hard to see that all symbol factors that can appear for $\mathrm{Li}_{n_1,...,n_d}(a_1, ..., a_d)$ are of the form a_i or $1 - \prod_{i=i_1}^{i_2} a_i$ with $1 \leq i_1 < i_2 \leq d$. So, e.g., for depth 2 we have

$$a_1,\ a_2,\ 1 - a_1,\ 1 - a_2,\ 1 - a_1 a_2 \tag{35}$$

as possible symbol factors. For depth 3 we have

$$a_1,\ a_2,\ a_3,\ 1 - a_1,\ 1 - a_2,\ 1 - a_3,\ 1 - a_1 a_2,\ 1 - a_2 a_3,\ 1 - a_1 a_2 a_3 \tag{36}$$

instead and for depth 4

$$a_1, a_2, a_3, a_4, 1 - a_1, 1 - a_2, 1 - a_3, 1 - a_4, 1 - a_1 a_2, 1 - a_2 a_3, 1 - a_3 a_4, 1 - a_1 a_2 a_3, 1 - a_2 a_3 a_4, 1 - a_1 a_2 a_3 a_4. \tag{37}$$

This motivates constructing the arguments for depth $d + 1$ inductively from the set of arguments for depth d. We will discuss this explicitly for the first few cases; it should be clear that this generalizes.

- Say, we have a list of all depth 1 arguments $r_1, r_2, ..., r_N$. By construction, r_i and $1 - r_i$ factorize over the alphabet. So we just have to look at all $1 - r_i r_j$ with $1 \leq i \leq j \leq N$ and try to factorize it over $\mathscr{L}$ using the Smith Decomposition and LLL-Algorithm as before. This yields a list of admissible tuples of arguments. Note that for (R_i, R_j) also (R_j, R_i) is an element of this list.

- Now we look for pairs of tuples $(R_1, R_2), (Q_1, Q_2)$ in the above list which fit together in the sense that $R_2 = Q_1$. Then, automatically, we have that the conditions that

  $$R_1, R_2, Q_2, 1 - R_1, 1 - R_2, 1 - Q_2, 1 - R_1 R_2, 1 - R_2 Q_1 \tag{38}$$

  shall factorize over the alphabet are met. Since in the first step we found all admissible tuples, this yields all triples (R_1, R_2, Q_2) satisfying above conditions. So we just have to test whether or not $1 - R_1 R_2 Q_2$ factorizes over the alphabet. If yes, add (R_1, R_2, Q_2) to the list of admissible triples.

- Now, look at triples (R_1, R_2, R_3) and (Q_1, Q_2, Q_3) which fit together if $R_2 = Q_1$ and $R_3 = Q_2$ and form in this case a candidate for an admissible quadruple (R_1, R_2, R_3, Q_3) such that only the factorization of $1 - R_1 R_2 R_3 Q_3$ has to be checked.

3 Symbology

3.1 Calculation of symbols, shuffles and projections

Recall that for $I(a_0;a_1,...,a_n;a_{n+1})$ with $G(a_1,...,a_n;a_{n+1}) = I(0;a_n,...,a_1;a_{n+1})$ the symbol is given recursively by

$$\mathcal{S}(I(a_0;a_1,...,a_n;a_{n+1})) = \sum_{j=1}^{n} \mathcal{S}(I(a_0;a_1,...,\hat{a_j},...,a_n;a_{n+1})) \otimes (a_{j+1}-a_j) - \mathcal{S}(I(a_0;a_1,...,\hat{a_j},...,a_n;a_{n+1})) \otimes (a_{j-1}-a_j) \tag{39}$$

and

$$\mathcal{S}(I(a_0;a_1;a_2)) = \otimes(a_2-a_1) - \otimes(a_0-a_1). \tag{40}$$

This structure lends itself well to an implementation via populating a tree whose nodes carry information about the current list of arguments $(a_0;a_1,...,\hat{a_j},...,a_n;a_{n+1})$ and the tuple (a_{j+1},a_j) or (a_{j-1},a_j) that has been singled out. At the end, all singled out tuples have to be collected and furnished with the correct signs. Taking the symbol is a linear operation (it even is an algebra homomorphism with respect to the shuffle product introduced next).

The shuffle relation provides a commutative multiplication in the usual sense yielding an algebra. Recall that the shuffle product (on words; extend by linearity and generalize to different algebras like the tensor algebra of symbols) is defined recursively via

$$(a_1 \cdots a_l) \bullet (b_1 \cdots b_r) = [(a_1 \cdots a_{l-1}) \bullet (b_1 \cdots b_r)]a_l + [(a_1 \cdots a_l) \bullet (b_1 \cdots b_{r-1})]b_r \tag{41}$$

and

$$\epsilon \bullet (a_1 \cdots a_l) = (a_1 \cdots a_l) \bullet \epsilon = a_1 \cdots a_l, \tag{42}$$

where ϵ denotes the empty word. This is implemented via a recursive function; sticking close to the above formulation.

On this algebra we define projection operators as follows:

$$\Pi_1 := \mathrm{id}, \quad \Pi_w(a_1 \otimes \cdots \otimes a_w) := \Pi_{w-1}(a_1 \otimes \cdots \otimes a_{w-1}) \otimes a_w - \Pi_{w-1}(a_2 \otimes \cdots \otimes a_w) \otimes a_1. \tag{43}$$

In the C++ code, this is implemented via constructing a suitable binary tree and assigning the correct signs. Both the shuffle as well as the projection operations are promoted to so-called λ-shuffles and λ-projectors, where $\lambda = (\lambda_1,...,\lambda_r)$ is an integer partition of $\mathbb{N} \ni w = \sum_i \lambda_i$:

$$\bullet_\lambda(a_1 \otimes \cdots \otimes a_w) := (a_1 \otimes \cdots \otimes a_{\lambda_1}) \bullet \cdots \bullet (a_{\lambda_1+\cdots+\lambda_{r-1}+1} \otimes \cdots \otimes a_w) \tag{44}$$

and

$$\Pi_\lambda := \Pi_{\lambda_1} \otimes \cdots \otimes \Pi_{\lambda_r}. \tag{45}$$

Since $\mathcal{S}$ is an algebra homomorphism, these λ-shuffles arise when we compute the symbol of products of G-functions; the λ_i correspond to the weights of the polylogarithms involved.

Those projection operators allow us to work up to products in the following sense. E.g., if we have, say, $f = G(a,b;x) \cdot G(c;y) \cdot G(d;z)$ then $\Pi_{\{4\}}(f) = \Pi_{\{3,1\}}(f) = \Pi_{\{2,2\}}(f) = 0$ but $\Pi_{\{2,1,1\}}(f) \neq 0$ and $\Pi_{\{1,1,1,1\}}(f) = \mathcal{S}(f) \neq 0$. Another example: if we want to work modulo products at weight n, then we can use $\Pi_{\{n\}}$.

3.2 Simplification of symbols

Using symbol calculus it is clear that simplification of symbols is tantamount to factorization of the symbol factors over the underlying alphabet (which, by construction of the ansatz functions to be explained in the next section, is always possible). Since polynomial rings over the integers or over fields are unique factorization domains, square-root-free expressions do not present huge problems. However, with square roots the picture becomes more complicated. The objective is here to remove as many square roots as possible. We discuss how the algorithm works with an example. Say, we want to simplify the symbol

$$12(a+\sqrt{b}) \otimes (c) \otimes (d+\sqrt{e}) + 6(a+\sqrt{b}) \otimes (c) \otimes (d-\sqrt{e}) + 6(a-\sqrt{b}) \otimes (c) \otimes (d+\sqrt{e}).$$

Of course, such a nice example is typically not apparent in the symbol at the beginning. One has to search for groups like this where simplification is possible (recall that we had assumed that $(A+\sqrt{B})\cdot(A-\sqrt{B})$ factorizes over the alphabet). After having found all such groups the algorithm first splits up the symbol in terms with equal numerical prefactors:

$$6(a+\sqrt{b})\otimes(c)\otimes(d+\sqrt{e})+6(a+\sqrt{b})\otimes(c)\otimes(d+\sqrt{e})+6(a+\sqrt{b})\otimes(c)\otimes(d-\sqrt{e})+6(a-\sqrt{b})\otimes(c)\otimes(d+\sqrt{e}).$$

Next, it groups the symbol into the largest group that can be simplified

$$6(a+\sqrt{b})\otimes(c)\otimes(d+\sqrt{e})+6(a+\sqrt{b})\otimes(c)\otimes(d-\sqrt{e})+6(a-\sqrt{b})\otimes(c)\otimes(d+\sqrt{e}).$$

and the rest

$$6(a+\sqrt{b})\otimes(c)\otimes(d+\sqrt{e}).$$

It determines, which term(s) to add for simplification to be achieved:

$$6(a+\sqrt{b})\otimes(c)\otimes(d+\sqrt{e})+6(a+\sqrt{b})\otimes(c)\otimes(d-\sqrt{e})+6(a-\sqrt{b})\otimes(c)\otimes(d+\sqrt{e})+6(a-\sqrt{b})\otimes(c)\otimes(d-\sqrt{e})=$$

$$=6(a^2-b)\otimes(c)\otimes(d^2-e).$$

The rest is changed accordingly:

$$6(a+\sqrt{b})\otimes(c)\otimes(d+\sqrt{e})-6(a-\sqrt{b})\otimes(c)\otimes(d-\sqrt{e}).$$

This whole procedure is repeated several times across the whole symbol. This way, the number of symbol letters used can be drastically reduced.

4 The Integration of Symbols algorithm

4.1 Basic idea

Given an (integrable) symbol s, we want to find a polylogarithmic function F such that $\mathcal{S}(F)=s$. This is done in three steps:

(i) Simplify the given symbol s using the procedures described above.

(ii) We are given a symbol alphabet meaning that we have a list of all irreducible letters (do not take the word irreducible too seriously when there are square roots in the game; in fact, it should be enough that the symbol alphabet is multiplicatively independent) appearing as factors in the given symbol s. This symbol alphabet satisfies the assumptions described in the first section.

The first step is to construct ansatz functions which we will use to fit the symbol. This consists of choosing basic function types (following Duhr, Gangl, Rhodes these are $\mathrm{Li}_1, \mathrm{Li}_2, \mathrm{Li}_3, \mathrm{Li}_4, \mathrm{Li}_{2,2}, \mathrm{Li}_5, \mathrm{Li}_{2,3}, \mathrm{Li}_6, \mathrm{Li}_{2,4}, \mathrm{Li}_{3,3}, \mathrm{Li}_{2,2,2}, ...$) and then constructing admissible arguments for them.

Important: It is our goal to not only find any function F that does the job, but it should be a particularly simple one. In light of this, it is clear that we should choose the simplest and fewest possible basic function types - these are the Li-functions with lowest depth for a given weight that still are able to span the entire function space defined by all the polylogarithms we are considering. Of course, it would be possible to just take all Li-functions into account, but this would defeat the purpose: if we were given a complicated amplitude in terms of G-functions of which we calculate the symbol s and then used all the Li-functions, we would merely rewrite the amplitude in terms of at least as complicated polylogs. Instead, we want some reduction of depth and henceforth reduction of complexity to take place.

(iii) Finally, we solve the linear system of equations over $\mathbb{Q}$ that arise from fitting the symbols of the ansatz functions to the given symbol. This is done successively using the projectors: roughly, we first work modulo any products, then work modulo products of two terms or more etc.

4.2 Results and problems for weight ≥ 5

I was able to achieve the following goals:

(i) Simplify huge symbols containing square root letters, thereby reducing the number of used letters.

(ii) Finding a minimal spanning set for 2dHPLs up to and including weight 4: this was done by using appropriate Lyndon words to generate a spanning set of 2dHPLs in terms G-functions, calculating their symbols, integrating those, collecting the ansatz functions used and reducing the resulting spanning set by trying to rewrite one of them in terms of the others.

While integration of the huge symbols did not succeed because of memory issues (which should be fixable with a little more effort), the problems encountered in finding a minimal spanning set for 2dHPLs for weight ≥ 5 are of a more fundamental nature, I believe.

Fact: For weight 5 there are 2dHPLs for which the procedure of rewriting them in terms of Li-functions by integrating the symbols as above does not work. In fact, the procedure fails for the overwhelming majority and works only for a few special cases. Up to and including weight 4 there are no complications whatsoever.

Example: for the symbol alphabet $x, y, 1-x, 1-y, 1-x-y, 1+y, x+y$ the following G-functions lead to a failure of the integration algorithm when using the basic function types as listed above:

$$G(0,0,0,1,1;x), G(0,0,0,1,-y;x), G(0,0,1,0,1;x), G(0,0,1,0,-y;x), G(0,0,1,1,1;x), G(0,0,1,1,-y;x), G(0,0,1,1,1-y;x), ... \tag{46}$$

The following worked:

$$G(0,0,0,1,1-y;x), G(0,0,0,-y,1;x), G(0,0,0,-y,-y;x), G(0,0,0,-y,1-y;x), G(0,1,1,1-y,1-y;x), ... \tag{47}$$

[*Adding the letter 2 does help a little but does not change the full picture. Neither do the other random letters I have tried to add.*]

Duhr, Gangl and Rhodes were aware of this, giving a maximal set of additional letters to add to the symbol alphabet in order to make it work:

- Problem 1: the maximal set of additional letters is quite large compared to the original set.
- Problem 2: The integration algorithm scales terrible with the number of letters - especially at higher weights.
- Problem 3: It is impossible to know beforehand which letters have to be added; one has to do a brute-force search. Because of Problem 2, one starts with single letters, then pair of letters, then triple of letters and so on. The number of possibilities grows factorially.

I think there should be a more structured approach and I will describe a possible way now. The starting point is to pin down exactly where the problem occurs for weight ≥ 5. I conjecture the following:

Claim: The depth reduction for Li-functions of weight ≥ 5 in terms of the Li-functions listed in 4.1 (ii) (and variants of those; e.g. using $\mathrm{Li}_{3,2}$ or $\mathrm{Li}_{4,1}$ instead of $\mathrm{Li}_{2,3}$) is only possible by using functions whose symbols contain factors which are not in the form $a, b, c, 1-a, 1-b, 1-c, 1-ab, 1-ac, 1-bc, 1-abc, ...$

Example: up to products, we have (see 2012.09840 or https://www.math.uni-hamburg.de/personen/charlton/talks/27_weight_4and5_mpls_slides.pdf):

$$\begin{aligned}
\mathrm{Li}_{3,2}(a,b) = &-3\mathrm{Li}_5(1-a)+5\mathrm{Li}_5(a)-6\mathrm{Li}_5\left(1-\frac{1}{ab}\right)+4\mathrm{Li}_5\left(-\frac{1}{b}\right)+4\mathrm{Li}_5\left(\frac{1}{b}\right)- \\
&-4\mathrm{Li}_5\left(\frac{1}{(a-1)b}\right)-\mathrm{Li}_5\left(\frac{1}{ab}\right)-\mathrm{Li}_5\left(\frac{b-1}{b}\right)+\mathrm{Li}_5\left(-\frac{(a-1)b}{b-1}\right)-8\mathrm{Li}_5\left(\frac{a}{1-ab}\right)- \\
&-3\mathrm{Li}_5\left(-\frac{1-ab}{(a-1)b}\right)-\mathrm{Li}_5\left(\frac{1-a}{1-ab}\right)+\mathrm{Li}_{4,1}\left(1,\frac{b-1}{b}\right)-\mathrm{Li}_{4,1}\left(1-a,\frac{1-b}{(a-1)b}\right)+ \\
&+\mathrm{Li}_{4,1}\left(\frac{1}{a},\frac{1}{b}\right)-\mathrm{Li}_{4,1}\left(-b,\frac{1}{1-b}\right)-\mathrm{Li}_{4,1}\left((a-1)b,\frac{a}{a-1}\right)+\mathrm{Li}_{4,1}\left((a-1)b,\frac{1}{1-b}\right)- \\
&-\mathrm{Li}_{4,1}\left(ab,\frac{1}{b}\right)-\mathrm{Li}_{4,1}\left(\frac{a}{1-ab},\frac{1}{a}\right)-\mathrm{Li}_{4,1}\left(\frac{a}{1-ab},1-b\right)+\mathrm{Li}_{4,1}\left(-\frac{(a-1)b}{1-ab},\frac{a}{a-1}\right)- \\
&-\mathrm{Li}_{4,1}\left(-\frac{ab}{1-ab},1-\frac{1}{ab}\right)-\mathrm{Li}_{4,1}\left(-\frac{ab}{1-ab},\frac{b-1}{b}\right).
\end{aligned} \tag{48}$$

Up to products means that computing the symbol of both sides and then taking Π_5 on both sides we get equal expressions. Note the following:

(i) When taking the symbol (but not modding out products), the left hand side only produces the letters $a, b, 1-a, 1-b, 1-ab$. The right hand side, however, produces additionally the letters $1+b, 1-ab+b, 1-ab-a$. These letters do not factorize over the alphabet (since by construction admissible arguments (R), (R,Q) have the property that $R, Q, 1-R, 1-Q, 1-RQ$ factorize over the alphabet); the algorithm cannot produce ansatz functions with arguments that produce such symbol letters. It does not seem to be the case that we can find another relation such that this does not happen. To this end, I have used the symbol alphabet $a, b, 1-a, 1-b, 1-ab$, constructed all admissible arguments from these letters and tried to run the integration algorithm with the goal of fitting the corresponding ansatz functions to the symbol of $\mathrm{Li}_{3,2}(a,b)$. This failed. Note that it worked, however, for *all* Li-functions of arbitrary depth up to and including weight 4. This seems to explain why the trouble starts at weight 5.

(ii) This is bad for the following reason: in the first step of the integration algorithm, we fit our ansatz functions to the given symbol modulo products. As I said before, modulo products LHS and RHS agree. Since the LHS only produces the symbol letters $a, b, 1-a, 1-b, 1-ab$ before and after Π_5, the RHS also can only produce the letters $a, b, 1-a, 1-b, 1-ab$ after Π_5. The evil letters therefore cancel out when applying Π_5. And they do so in a very specific and simple way:

- Evil letter $1+b$: It is produced by the following group:

$$\mathfrak{g}_1 = 4\mathrm{Li}_5\left(-\frac{1}{b}\right) - \mathrm{Li}_{4,1}\left(-b, \frac{1}{1-b}\right) \tag{49}$$

  We have $\mathscr{S}(\mathfrak{g}_1) \supset \{b, 1-b, 1+b\}$ but $\Pi_5(\mathscr{S}(\mathfrak{g}_1)) \supset \{b, 1-b\}$, so the evil letter cancels.
- Evil letter $1-ab+b$: It is produced by the following group:

$$\mathfrak{g}_2 = -4\mathrm{Li}_5\left(\frac{1}{(a-1)b}\right) - \mathrm{Li}_{4,1}\left((a-1)b, \frac{a}{a-1}\right) + \mathrm{Li}_{4,1}\left((a-1)b, \frac{1}{1-b}\right). \tag{50}$$

  We have $\mathscr{S}(\mathfrak{g}_2) \supset \{a, b, 1-a, 1-b, 1-ab, 1-ab+b\}$ but $\Pi_5(\mathscr{S}(\mathfrak{g}_2)) \supset \{a, b, 1-a, 1-b, 1-ab\}$, so the evil letter cancels.
- Evil letter $1-ab-a$: It is produced by the following group:

$$\mathfrak{g}_3 = -8\mathrm{Li}_5\left(\frac{a}{1-ab}\right) - \mathrm{Li}_{4,1}\left(\frac{a}{1-ab}, \frac{1}{a}\right) - \mathrm{Li}_{4,1}\left(\frac{a}{1-ab}, 1-b\right). \tag{51}$$

  We have $\mathscr{S}(\mathfrak{g}_3) \supset \{a, b, 1-a, 1-b, 1-ab, 1-ab-a\}$ but $\Pi_5(\mathscr{S}(\mathfrak{g}_5)) \supset \{a, b, 1-a, 1-b, 1-ab\}$, so the evil letter cancels.

  Note: the groups are disjoint and produce only one evil letter each. This might facilitate the search for them. Similar results also hold for the second example given in the talk linked above (i.e. for the depth reduction of $I_{3,1,1}$). Of course, these are only empirical observations at the moment.

  I want to stress again: although in the first step of the integration algorithm everything is done up to products only and under Π_5 the RHS produces just good symbol letters, the fact that - first - the symbols of the groups above come from functions where the evil letters had to non-trivially cancel out when applying Π_5 and - second - those groups cannot be rewritten in terms of polylogs whose mere symbol (i.e. without projecting) just produces good letters seems to imply that the first step of the algorithm has to fail.

So, the step forward seems to be to find a way of constructing such groups with the following properties:

(i) They contain functions whose mere symbol contains evil letters.

(ii) After applying Π_5, the evil letters cancel out non-trivially.

If we can construct these groups, we can do the first integration step since we then have access to more symbols with the correct letters that were previously out of reach. But the fact, that we now have introduced functions which produce symbol letters that are not factorizable over the given symbol alphabet produces further problems

for the subsequent integration steps. This is because after we have found a function F_5 that has the property that $\Pi_5(s) = \Pi_5(\mathscr{S}(F_5))$ (i.e. after the first step), we update $s \leftarrow s - \mathscr{S}(F_5)$ leading to the problem that now s contains evil letters. We can remedy this as follows: look at which evil letters actually appear in the updated s; i.e. make a list of all those symbol letters that cannot be factored over the current alphabet. Reduce those symbol letters into irreducibles (with the usual caveat for square roots). Add the irreducibles to our symbol alphabet. The hope is, that one does not have to add too many new letters but only one or two or maybe three. I do not currently know whether or not this is too optimistic. For the rest of the algorithm we only consider (products of) polylogs of weight ≤ 4 which means that, once we have updated the symbol alphabet and recomputed the ansatz functions I do not anticipate problems. Note that this seems to effectively solve the problem of brute-forcing the additional letters described in Duhr, Gangl, Rhodes. While I have described the problem here only for weight 5, for weight 6 (and, indeed, higher weights) similar reasoning applies. Additionally, e.g. for the $\Pi_{5,1}$-step, the groups found at weight 5 have to be recycled.