Z-Polyhedra and LBLs in PolyLib

Vincent Loechner
Dhimiter Riza
ICube, Université de Strasbourg, and Inria CAMUS
Illkirch, France

Abstract

Z-polyhedra and linearly bounded lattices (LBLs) provide expressive representations for sets of integer points beyond classical rational or integer polyhedra. Although **Z**-polyhedra were introduced in PolyLib more than two decades ago, their implementation was limited and did not support robust manipulation of general unions of **Z**-polyhedra or LBLs.

In this paper, we present a sound, complete and unified implementation of **Z**-polyhedra, LBLs, and their finite unions in PolyLib. Our approach relies on the representation of an LBL as the affine lattice function image of a coordinate polyhedron. A normalization scheme enables union, intersection, difference, image, preimage, inclusion testing, and conversion of a union of LBLs to a union of Z-domains. Our implementation is using efficient algorithms whenever possible but is also able to compute the most complex by nature cases, carefully handling lattice normalization, equality elimination and holes. It has been validated on a wide range of test cases.

This work significantly extends PolyLib's capabilities and makes advanced lattice-based representations practically usable in polyhedral compilation and related applications.

ACM Reference Format:
Vincent Loechner and Dhimiter Riza. January, 28, 2026. **Z**-Polyhedra and LBLs in PolyLib. In *Proceedings of 16th International Workshop on Polyhedral Compilation Techniques (IMPACT '26).* ACM, New York, NY, USA, 8 pages.

1 Introduction

The polyhedral library (or PolyLib) [6, 10] is a library that was first developed in the 1990's to manipulate finite unions of rational polyhedra based on the double description of Motzkin. **Z**-polyhedra were added to PolyLib in 2000 [7] using the algorithms introduced by Quinton, Rajopadhye and Risset [9] that represent a **Z**-polyhedron as the intersection of an integer lattice and a polyhedron. Later works by Gautam [1, 2] used a representation of a **Z**-polyhedron close to the one of an LBL, as the affine image by a full column Hermite-normal form (HNF) matrix of a full-dimensional integer polyhedron (the *coordinate polyhedron*):

$$\mathcal{Z} = \left\{ Lx + l \;\middle|\; Cx + c \geq 0,\ x \in \mathbb{Z}^d \right\}$$

This is possible since **Z**-polyhedra is a subclass of LBLs: every **Z**-polyhedron can be represented as an LBL. However, computing a full-dimensional integer coordinate polyhedron of any LBL is NP-hard, as pointed out by Iooss and Rajopadhye [3]. In their work, Iooss and Rajopadhye used the same representation of a **Z**-polyhedron as an image by a lattice function, but they dropped the condition to be canonical that the integer coordinate polyhedron should be full-dimensional, at the price of more complex comparison and image algorithms.

We used some of those ideas to implement the complete set of operations on unions of LBLs and **Z**-polyhedra in PolyLib. We extended Gautam's work to handle some unforeseen issues; extended Iooss and Rajopadhye's canonical form definition; and implemented the existential variable elimination to transform an LBL into a union of **Z**-polyhedra. The main functions available to the user are: `LBLIntersection`, `LBLUnion`, `LBLIncluded`, `LBLDifference`, `LBLImage`, `LBLPreimage`, `LBL2ZDomain`, and `LBLSimplifyEmpty`.

The normalization of a single LBL, presented in section 4, is the cornerstone that allows easy implementation of those basic functions. It sets the lattice matrix to affine Hermite normal form and eliminates the explicit equalities of the coordinate polyhedron, but keeps the columns of zeros in the lattice function that are necessary to preserve the "holes": it only eliminates the columns of zeros when the sufficient condition to eliminate a dimension that the dark shadow is equal to the exact shadow is verified, as shown by Pugh [8].

The transformation of an LBL into a union of **Z**-polyhedra (function `LBL2ZDomain`) further eliminates all existential variables and all zero columns of the lattice. However, this operation is known to be very costly in the worst case, in computation time and in output size. For each dimension that has to be eliminated, when the dark shadow does not cover the exact shadow, we need to explicitly compute the union of points that are lying in the difference between the exact shadow and the dark shadow and that are not holes.

This paper first summarize related work in section 2, then recalls some mathematical background in section 3. We present our representation of normal LBLs and their unions in section 4, and section 5 details the algorithms implemented in the PolyLib new operations on lattices, single LBLs and union of LBLs.

IMPACT '26, 2026, Krakow, Poland
January, 28, 2026.

2 Related work

In 1991, Pugh [8] implemented a fast and practical integer linear programming (ILP) solver known as the Omega test. It allows one to decide whether a set of constraints containing integer existential variables admits an integer solution. He also proposed a sufficient condition for the fast elimination of an existential variable: if the exact (rational) shadow of the eliminated variable is covered by its dark shadow —a thick-face-like projection that guarantees that all projected points have at least one integer preimage— then the variable can be removed. We reuse this idea to eliminate existential variables and reduce the dimensionality of the coordinate polyhedra in our representation.

In 1997, Quinton, Rajopadhye and Risset [9] proposed a set of algorithms to handle **Z**-polyhedra, represented as the intersection of an integer lattice and a polyhedron. These algorithms were implemented in PolyLib in 2000 [7], but this implementation suffers from several limitations: only unimodular transformations were supported, existential variable elimination was not available, general LBLs were not handled, and the implementation is no longer maintained.

Gautam and Rajopadhye [1, 2] proposed a set of algorithms that handle general LBLs, with **Z**-polyhedra as a subclass of this more general class of objects, using a unified representation as the image by an affine lattice function of an integer polyhedron. However, this theoretical work was not implemented, and some of the proposed algorithms exhibit unforeseen issues.

Iooss and Rajopadhye [3] implemented the LBLs using the same representation as Gautam in a Java library included in AlphaZ. Our work is the closest to this approach, however, this implementation appears no longer maintained or distributed.

Building on these prior efforts, our work now provides an efficient, general and unified implementation of Z-polyhedra, LBLs, and their unions, now broadly available to users through its integration into PolyLib [5].

3 Mathematical Background

3.1 Integer Affine Function and Lattice

An **integer affine function** or *lattice function* is represented as an integer matrix M and a constant integer vector m. The affine lattice generated by such a function is a subset of $\mathbb{Z}^n$ and follows the form:

$$\mathcal{A} = \left\{ Mz + m \;\middle|\; z \in \mathbb{Z}^d \right\}$$

If the columns of matrix M are linearly independent they constitute a basis of the generated lattice, and m is its offset.

To be in canonical form M and m have to satisfy the condition that the matrix

$$\hat{M} = \begin{pmatrix} 1 & 0 \cdots 0 \\ m & M \end{pmatrix}$$

is in column left Hermite normal form (HNF): it is lower triangular, and all elements on the left of the *pivots* (the first nonzero entries of the columns) are in the interval $[0, \text{pivot}[$. Any matrix generating a lattice can be transformed into its unique HNF, which is unimodular-equivalent to the original matrix.

Notice that the canonical matrix $\hat{M}$ is not necessarily square: it can have more rows than columns, in which case the function spreads points to a higher dimension target space than the origin space. Also notice that $\hat{M}$ might contain columns of zeros on its right. This happens when a dimension of the origin space is eliminated by the function; since matrix $\hat{M}$ is lower triangular, those zero columns are necessarily on the right of the matrix.

The canonical form ensures uniqueness of the non zero columns of $\hat{M}$: due to the properties of the HNF, two functions spreading the same lattice yield the same lattice matrix $\hat{M}$, except for the presence of an arbitrary number of zero columns on their right.

3.2 Z-Polyhedron and LBL

In the following definitions, M and C are integer matrices, and m and c are integer vectors.

An **integer polyhedron** is the set of all points of $\mathbb{Z}^d$ verifying a finite set of affine inequalities:

$$\mathcal{P} = \left\{ z \in \mathbb{Z}^d \;\middle|\; Cz + c \geq 0 \right\}.$$

A **Z-polyhedron** is the intersection of an integer lattice and an integer polyhedron:

$$\mathcal{Z} = \{ z \in \mathbb{Z}^d \mid \exists z' \in \mathbb{Z}^{d'},\ z = Mz' + m,\ Cz + c \geq 0 \}.$$

A simple **LBL**[1] is the affine integer image by a lattice function of an integer polyhedron, called the coordinate polyhedron:

$$\mathcal{L} = \left\{ z = My + m \;\middle|\; Cy + c \geq 0,\ y \in \mathbb{Z}^d \right\}.$$

An LBL can be represented in normalized form as a pair: a canonical affine function as defined in the previous subsection; and a *rational* coordinate polyhedron which does not contain any implicit equality. The points y are the integer points contained in this rational polyhedron. Notice that the corresponding *integer* coordinate polyhedron might contain equalities, but it is an NP-hard problem to find those implicit integer equalities from an arbitrary rational polyhedron [3], so we do not guarantee their absence. This implies however that two different LBLs in normalized form may represent the same set of integer points, and the only way to decide for equality between such two sets is to explicitly verify their mutual inclusion or to check the emptiness of their difference.

[1] LBL is the commonly used acronym of "linearly bounded lattice", which it is not: an LBL is *not* a lattice bounded by linear inequalities, that is the definition of a **Z**-polyhedron.

3.3 Union of Z-polyhedra and LBLs

Based on the works of Le Verge [4] and Gautam [1], we know that the family of integer polyhedra is strictly contained in the family of **Z**-polyhedra which is itself strictly contained in the family of LBLs. Gautam also showed that any **Z**-polyhedron can be represented as an LBL, and that any LBL can be represented as a finite union of **Z**-polyhedra. Also, a union of LBLs or **Z**-polyhedra can not necessarily be represented as a single LBL: two members of the union can have two different full-dimensional lattices whose union is not a lattice. So we have:

$$\begin{aligned}&\text{integer polyhedra} \subset \textbf{Z}\text{-polyhedra} \subset \text{LBLs}\\ &\subset (\text{union of } \textbf{Z}\text{-polyhedra}) = (\text{union of LBLs})\end{aligned}$$

In the following, we call a **Z-domain** a finite union of **Z**-polyhedra. We call a **single LBL** an LBL having one single lattice, but possibly associated to multiple polyhedra (a polyhedral domain). To simplify the reading, we call **LBL** a union of single LBLs.

In our implementation, the same internal representation is used for **Z**-domains and LBLs, as a list of pairs: (lattice matrix, rational polyhedral domain). The only difference between a **Z**-domain and an LBL is that there are no zero columns in the lattice matrices of a **Z**-domain. In the normalized form of an LBL (a union of single LBLs), each lattice matrix is canonical and it appears at most once in this list. Notice that, as for simple LBLs, this normalized form of an LBL does not guarantee uniqueness (1) for the same reason as above since the union is composed of simple LBLs, and moreover (2) since a given integer set can be represented using several different lattice decompositions of different dimensions. It is a hard problem to unify arbitrary lattice unions in a canonical form, so we decided to leave the complexity of the comparison and difference algorithms in those functions when called explicitly, but still do a fast check for obvious inclusion first.

4 Normalized Representation

4.1 Single LBL Normalized Form

The algorithm to transform an arbitrary single-lattice LBL into its normalized form has three main steps: lattice function normalization, elimination of the equalities in the domain, and elimination of the non-necessary zero columns of the lattice matrix. In the following, we call L the *homogeneous* lattice matrix, as denoted by $\hat{M}$ in section 3.1: the constant dimension is included as the first column of the matrix, and the extra first row represents the homogeneous coordinate. We call D the homogeneous polyhedral domain, and x and z also represent vectors in homogeneous dimension.

Example 1. *The following example will be used along this subsection to illustrate each step of the normalization algorithm. Let:*

$$\mathcal{L}_0 = \left\{ \begin{pmatrix}1 & 1 & 0\end{pmatrix}\begin{pmatrix}i\\j\\k\end{pmatrix} \;\middle|\; i = 2j, 0 \le j \le 5, \begin{pmatrix}i\\j\\k\end{pmatrix} \in \mathbb{Z}^3 \right\}$$

In homogeneous dimensions the lattice matrix is:

$$L = \begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 1 & 0\end{pmatrix}$$

Step 1. Canonicalize the lattice matrix. We first compute the left Hermite normal form of matrix L and its associated unimodular matrix U as:

$$H = L\,U$$

The canonical homogeneous lattice function is simply H. But in order to spread the same LBL points the domain has to be transformed. Since the original LBL is the set of points:

$$z = Lx, x \in D$$

we can rewrite:

$$z = H\,x' = L\,U\,x', \text{ with } x' = U^{-1}\,x, x \in D$$

So, for matrix H to be used as lattice matrix, the domain has to be transformed by homogeneous matrix U^{-1} to contain the points $x' = U^{-1}\,x$. Since matrix U is unimodular, this transformation is always valid: it maps each integer point of the original coordinate domain to an integer point of the domain transformed by the preimage by U.

Example 1 (continued)**.** *The HNF of L is computed as:*

$$L = H\,U = \begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\end{pmatrix}\begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & -1 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{pmatrix}$$

So H represents the new lattice matrix, and the domain is transformed by the preimage function by U. The resulting LBL, where the lattice matrix is in HNF, is:

$$\mathcal{L}_1 = \left\{ \begin{pmatrix}1 & 0 & 0\end{pmatrix}\begin{pmatrix}i\\j\\k\end{pmatrix} \;\middle|\; i = 3j, 0 \le j \le 5, \begin{pmatrix}i\\j\\k\end{pmatrix} \in \mathbb{Z}^3 \right\}$$

Step 2. Eliminate equalities. The domain D is first simplified by a call to the PolyLib core function `DomainConstraintSimplify`: any inequality has its constant rounded to a multiple of the *gcd* of its coefficients and an equality without an integer solution just eliminates the polyhedron. If a polyhedron becomes rational-empty, it is removed from the domain.

Then, in order to eliminate equalities from a domain D, we first scan the polyhedra of the union and separate them into subdomains composed of polyhedra having identical sets of equalities. They will be separated as different single LBLs in the list of LBLs composing a union of LBLs. Let us call

each of these subdomain S, and the equalities that it verifies homogeneous matrix E.

Then, we compute $K = \ker(E)$ using HNF: we compute $E\,U = H$, and we know that: $E \ker(E) = 0$, so, since the right columns of H are composed of zeros, the corresponding right columns of matrix U form a basis of the kernel of E.

The HNF of K, called J, is the integral kernel of the matrix of equalities. As a consequence it can be used to reduce the lattice L and the domain S to remove its equalities. The LBL:

$$z = L\,x,\ x \in S$$

is transformed into:

$$z = L\,J\,x',\ \text{with } J\,x' = x,\ x \in S$$

So the new lattice generating the same set of integer points is $L\,J$, and the new domain is the preimage of S by J, that is without any equality.

Notice that matrix J is not necessarily unimodular nor invertible, however this transformation is valid since it will eliminate some dimensions and eliminate some rational points that verify the equalities, but not the integer ones because J is the integral kernel of E.

Example 1 (continued)**.** *The homogeneous equality matrix is:*

$$E = \begin{pmatrix} 0 & 1 & -3 & 0 \end{pmatrix}$$

Its integral homogeneous kernel is computed and we get:

$$J = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Finally, the new homogeneous lattice matrix is:

$$L' = L\,J$$

and we compute the preimage of the coordinate polyhedron by J. The resulting LBL is as follows, with the i dimension used in the definition of $\mathcal{L}_1$ eliminated:

$$\mathcal{L}_2 = \left\{ \begin{pmatrix} 3 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \;\middle|\; 0 \le i \le 5, \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{Z}^2 \right\}$$

Step 3. Eliminate zero columns. This step consists of a loop on the zero columns of L, and remove the corresponding dimension only if the sufficient condition described below is verified. If some columns are eliminated by the loop, the remaining columns are scanned again to ensure that no more elimination is possible. Although this algorithm does not guarantee that the final result is unique, since a different elimination order could generate a different result, it reaches a fixed point and proved to be efficient in experimental cases.

We eliminate a column of zeros of lattice L only when we can ensure the sufficient condition that the dark shadow covers the exact shadow of the domain. As defined by Pugh [8], the exact shadow is the (rational) projection of D along the considered dimension. The dark shadow is the projection of D *restricted to its inside*, ensuring that at least one integer point of the origin domain maps to the projection, so that it is not a *hole*: this is done by adding the coefficient of the eliminated dimension minus one to the constant of each positive constraint on the eliminated dimension. If the dark shadow covers the exact shadow then there are no holes and this dimension can be eliminated: we remove the corresponding column in L and project the domain along this dimension.

Example 1 (continued)**.** *There is a remaining zero column on the right of the lattice defining $\mathcal{L}_2$. We compute the exact and the dark shadow of the coordinate polyhedron along this dimension and get the two identical polyhedra:*

$$E = D = \{i \mid 0 \le i \le 5\}$$

So this dimension can be integer-eliminated by simple projection along j, and we get the final normalized LBL, which is a $\mathbf{Z}$*-polyhedron:*

$$\mathcal{L}_0 = \mathcal{L}_1 = \mathcal{L}_2 = \mathcal{L}_3 = \{\, 3i \mid\ 0 \le i \le 5, i \in \mathbb{Z}\}$$

4.2 LBL Union Normalized Form

An LBL is a list of single LBLs. Its normalized form is such that:

1. each single LBL composing the list has been transformed into its normalized form by the above method;
2. there are no empty LBLs in the list, unless the whole list is a single empty LBL (in which case the domain is empty, but the lattice matrix height determines the dimension of the space that this empty LBL lies in);
3. each lattice matrix appears at most once in the list. The algorithm to ensure this condition scans all pairs of single LBLs of the list and if it finds two identical lattice matrices, it merges the single LBLs together by computing the union of their associated polyhedral domains.

5 Operations

5.1 LBL Type

In our implementation in PolyLib the following structure represents an LBL:

```
typedef struct lbl {
  Matrix *Lat;
  Polyhedron *P;
  struct lbl *next;
} LBL;
```

with `P` a polyhedral domain (a union of polyhedra): it uses the PolyLib representation of a rational polyhedral domain; matrix `Lat` is a homogeneous PolyLib matrix, of the form:

$$\texttt{Lat} = \begin{pmatrix} M & m \\ 0 & 1 \end{pmatrix}$$

and `next` is used to link single LBLs in a union.

The same LBL data structure is used to represent single LBLs, LBL unions, **Z**-polyhedra and **Z**-domains. To distinguish when we are dealing with a single LBL or with a (union of) LBLs we have explicitly named the functions manipulating them, e.g. `sLBLImage()` computes the image of a single LBL, while `LBLImage()` computes the image of an LBL union. The only functions that are exposed to the user of the library are the functions manipulating LBL unions.

The empty LBL of dimension d is represented as a `Lat` matrix having $d+1$ rows, and a `NULL` domain.

5.2 Operations on Lattices

The basic functions to manipulate lattices are:

```
Bool isNormalLattice   (Matrix *A);
Bool isEmptyLattice    (Matrix *A);
int  LatCountZeroCols  (Matrix* A);
Bool LatticeIncluded   (Matrix *A, Matrix *B);
Bool isEqualLattice    (Matrix *A, Matrix *B);
Bool isSameLatticeSpace(Matrix *A, Matrix *B);
```

Function `LatticeIncluded(A, B)` checks if `A` is included in `B`. Function `isEqualLattice(A, B)` checks if `A` is exactly equal to `B`, while `isSameLatticeSpace(A, B)` checks if `A` and `B` are equal ignoring their zero columns. Those functions implement pretty simple algorithms. Three more advanced functions are explained in the following:

a) Affine Hermite normal form

```
void AffineHermite(Matrix *A, Matrix **H,Matrix **U)
```

Compute `H` and `U` such that: `A = H U`, `H` is the affine left Hermite normal form of `A` and `U` is unimodular. This function first puts the last row and column of `A` as first row and column, then calls the PolyLib left Hermite basic engine, and finally puts the first row/column back to last in the matrices `A`, `H` and `U`. `U` can be left to `NULL` if not used be caller.

b) Intersection

```
Matrix *LatticeIntersection(Matrix *A, Matrix *B)
```

Let us call $\begin{pmatrix} A & a \\ 0 & 1 \end{pmatrix}$ = A and $\begin{pmatrix} B & b \\ 0 & 1 \end{pmatrix}$ = B.
The intersection function first builds matrix:

$$T = \left(\begin{array}{cc|cc} 1 & 0\ldots0 & 1 & 0\ldots0 \\ a & A & b & B \\ \hline 1 & 0\ldots0 & 0 & 0\ldots0 \\ a & A & 0 & 0\ldots0 \end{array}\right)$$

Then, the left Hermite normal form of this matrix is:

$$\mathrm{HNF}(T) = \left(\begin{array}{cc|cc} \multirow{2}{*}{D} & & 0 & 0\ldots0 \\ & & 0 & 0\ldots0 \\ \hline \multirow{2}{*}{X} & & 1 & 0\ldots0 \\ & & j & J \end{array}\right)$$

where $\mathtt{J} = \begin{pmatrix} J & j \\ 0 & 1 \end{pmatrix}$ is by construction the intersection of the two lattices. The reason this works is because HNF produces zeros in the upper right block of the matrix, and the corresponding columns in the bottom-right block are made up of a linear combination of the vectors from `A` and `B`. Since HNF(T) spreads the same points as the original matrix T, the bottom right part is necessarily composed of the set of vectors spreading the same points as `A` and as `B`: they are both in `A` and in `B`, so they are the intersection of `A` and `B`.

If, after computation of the HNF, the coefficient above vector j is not equal to one then the integer intersection is empty.

c) Difference

```
LatticeUnion *LatticeDifference(Matrix *A,Matrix *B)
```

In general, the difference between two lattices is a non-finite union of lattices. It is finite if the dimensions of `A` and `B` are equal: this function requires `A` and `B` to have the same number of rows and non-zero columns. The algorithm that we implemented first computes the intersection `J` = `A` ∩ `B`, and then takes `J` out of `A`: it builds the matrix spreading all points of `A` that are not part of `J`.

We define the *variants* of a pivot row of `A` as the rows that spread the points that are part of `A` but not part of `J`. To generate all lattices that spread points that are part of `A` but not part of `J`, we scan all pivots of `A` and all variants of the row containing the pivot are added to the result, along with the above rows coming from `J`, and the rows below coming from `A` (constant adjusted as required, if the pivot changes). In the end the result is the union of all those lattices, that are by construction spreading all the points of `A` that are not part of `J`.

For example, if `A` contains row (2 1) and `J` contains row (24 3), we would want to generate all rows: (24 1), ~~(24 3)~~, (24 5), (24 7), ... (24 21), (24 23) that spread all points that lie in `A` but not in `J`. This enumeration of the variants (there are 12 here) can be optimized by computing the prime divisors of the ratio of the pivots (24/2) and assembling some of those generating rows together. For example, the set of rows (24 1), (24 5), (24 9), (24 13), (24 17), (24 21) can be represented as the single row (4 1) generating the same set of points!

Taking this property into account, our general algorithm for generating all variants of a pivot row is as follows. The ratio (always integer) between the pivot of `J` and the corresponding pivot p of `A` is decomposed into its prime divisors d_i. Then the row of `J` is transformed: the pivot is replaced by each possible value $p * d_i^x$, and the constant by all possible values with a step of $p * d_i^{x-1}$, with x enumerating the number of identical divisors in the decomposition. If such a generated row hits the intersection row, it is not added to the result. All the others, that spread points that do not hit the intersection, are added to the result.

In our previous example, A contains row (2 1) and J row (24 3), our algorithm will compute the prime divisors of $24/2 = 12$, that are 2, 2 and 3, and then generate the following rows. For $2 * 2^1$: (4 1), ~~(4 3)~~; for $2 * 2^2$: ~~(8 3)~~, (8 7); for $2 * 3$: (6 1), ~~(6 3)~~, (6 5). Then it takes out the struck out ones since they hit (24 3). The decomposition in prime divisors guarantees that the minimal number of variants of the row are generated (there are only 4 in the end, instead of 12), but still that all possible variants of the row of A are generated.

5.3 Operations on Single LBLs

The following operations on single LBLs have been implemented in PolyLib. Recall that a single LBL contains one single lattice function, but is associated to a domain that may be a union of several polyhedra. Those functions are the core of the library for manipulating unions of LBLs, presented in the next subsection.

a) Normalized form

```
void sLBLCanonical(LBL *A)
```

Compute the normalized form of a single LBL (in place: modifies A), as described in section 4.1.

b) Image

```
LBL *sLBLImage(LBL *A, Matrix *M)
```

Let A = {L, D}. The image function is pretty simple: the set of points $x' = Mx$ is generated by the LBL {M L, D}. We compute the matrix product M L and normalize the resulting single LBL.

c) Preimage

```
LBL *sLBLPreimage(LBL *A, Matrix *M)
```

If matrix M is integer invertible, we just compute the single LBL image by M^{-1}, as explained in the previous function.

The preimage function is more complex in the other case, when M is not invertible. We need to compute the set of points x' that verify $Mx' + m = Lx + l$, with $x \in D$, and $M = \begin{pmatrix} M & m \\ 0 & 1 \end{pmatrix}$, $L = \begin{pmatrix} L & l \\ 0 & 1 \end{pmatrix}$. To do so, we explicitly build the following single LBL:

$$\left\{ \begin{pmatrix} \mathrm{Id} & 0\ldots0 & 0 \\ 0 & 0\ldots0 & 1 \end{pmatrix} \begin{pmatrix} x' \\ x \\ cst \end{pmatrix} \middle| Mx' + m = Lx + l, x \in D \right\}$$

So domain D is extended to the dimension of x plus x' and the equalities are added to the domain; the lattice is as given in the above equation. Then this single LBL is normalized, the equalities are eliminated, and we get the result.

d) Intersection

```
LBL *sLBLIntersection(LBL *A, LBL *B)
```

If the input LBLs are **Z**-domains (their lattices contain no zero columns), then the hulls of A and B are guaranteed to contain no hole, and we can use the following simple algorithm:

- compute J, the intersection of L_A and L_B (the lattices of A and B).
- if J is empty: return the empty LBL;
  else: return the LBL {J, D}, where D is the preimage by J of the intersection of the hulls of A and B:

$$D = J^{-1}(L_A D_A \cap L_B D_B)$$

In the other case, if A or B is an LBL possibly containing holes, we need to build the resulting LBL explicitly similarly to the previous function. We construct the following LBL, extending the dimension of the domains to their sum and adding the equalities, associated to the lattice selecting one of them:

$$\left\{ \begin{pmatrix} 0\ldots0 & L_A & l_A \\ 0\ldots0 & 0 & 1 \end{pmatrix} \begin{pmatrix} z' \\ z \\ cst \end{pmatrix} \middle| \begin{array}{c} L_A z + l_A = L_B z' + l_B, \\ z \in D_A,\ z' \in D_B \end{array} \right\}$$

then this LBL is normalized, and we get the result.

e) Compute holes

```
Polyhedron *sLBLCompute_holes(LBL *A)
```

This function computes the union of polyhedra containing the holes of the single LBL A. It is required by the next two functions (sLBLComplement and sLBL2ZDomain) in case they handle LBLs with holes; when A is a **Z**-domain, it immediately returns an empty domain.

The algorithm consists of a scan of possible holes in the projection of the coordinate polyhedron D, and an explicit search for all integer feasible solution, as follows:

- compute the domain difference R = (exact shadow) \ (dark shadow) of D,
- compute Q = expand R to the dimension of D and intersect it with D,
- scan R (or the outer dimensions of Q) and for each vector r of R:
  - scan the inner dimensions of Q, and as soon as an integer solution is found exit this scan, after adding the integer vector r to a domain called not_a_hole.

Finally, return Universe(dim(R)) - not_a_hole.

f) Complement

```
LBL *sLBLComplement(LBL *A)
```

In general, the complement of a single LBL is a union of LBLs. Let A = {L, D}. This function builds the union of: the points that are not part of the hull of A + the points that do not lie in lattice L + the holes of A, as:

1. the single LBL:

$$\{z \in Z^d | z \in DomainComplement(hull(\mathtt{A}))\}$$

The hull of A is the polyhedral domain: image by function L of domain D.

2. the union of LBLs:

   $\{Mz | z \in Z^d\}$ for each M in `LatticeDifference`$(Z^d, \mathtt{L})$

   Since the difference between two lattices is a union of lattices, this builds a union of LBLs. The coordinate polyhedron could have been chosen as the preimage of the hull of A, but it is unnecessarily more complicated to do so.

3. the LBL:

   $$\{\mathtt{L}_\emptyset z | z \in \mathtt{sLBLCompute\_holes(A)}\}$$

   where $\mathtt{L}_\emptyset$ is L without its zero columns.

g) LBL to Z-domain

```
LBL *sLBL2ZDomain(LBL *A)
```

This function just builds the LBL:

$\{\mathtt{L}_\emptyset z | z \in \mathtt{DomainDifference}(E,\ \mathtt{sLBLCompute\_holes(A)})\}$

where E is the exact shadow of A and $\mathtt{L}_\emptyset$ is L without its zero columns.

5.4 Operations on LBL Unions

The basic operations to allocate and manipulate an LBL are:

```
LBL *LBLAlloc    (Matrix *Lat, Polyhedron *Domain);
void LBLFree     (LBL *A);
void LBLPrint    (FILE *fp, char *format, LBL *A);
LBL *LBLCopy     (LBL *A);
LBL *EmptyLBL    (int dimension);
LBL *UniverseLBL (int dimension);
Bool isEmptyLBL  (LBL *A);
```

The following functions perform advanced operations on LBLs:

a) Normalized form

```
void LBLCanonical(LBL *A)
```

Compute the normalized form of an LBL, in place, as described in section 4.2. All the functions given below make a call to this function after their operation.

b) Union

```
LBL *LBLUnion(A, B)
```

Link together a copy of A and of B.

c) Image

```
LBL *LBLImage(A, M)
```

Build the union of the images by matrix M of each single LBL of A.

d) Preimage

```
LBL *LBLPreimage(A, M)
```

Build the union of the preimages by matrix M of each single LBL of A.

e) Intersection

```
LBL *LBLIntersection(A, B)
```

Build the union of all intersections between pairs of the single LBLs of the list A and of the list B.

f) Complement

```
LBL *LBLComplement(A)
```

Build the intersection of the complements of each single LBL composing A.

g) Difference

```
LBL *LBLDifference(A, B)
```

First compute the intersection $I = \mathtt{A} \cap \mathtt{B}$. If I is empty, the result is A. If I is equal to A then B covers A and the result is empty.

If those simple tests fails, the general case computes the difference between A and all single LBLs composing I, by successively intersecting A with the complement of each single LBL composing I.

h) Simplification

```
void LBLSimplifyEmpty(A)
```

Scan all domains stored in A and remove the polyhedra that have no integer solution, in place.

This function first checks if a polyhedron has a ray, a line, or an integer vertex. If found, it has an integer solution (since the constraints have been simplified, the presence of a ray or line ensures that an integer solution exists).

If not found, a systematic search scans the integer points of the polyhedron in lexicographic order and early exits if a solution is found. In case an integer solution is found, a constraint is added to the outermost dimension of the polyhedron such that a subsequent call, or the removal of holes on the same polyhedron, will hit this integer solution when scanning its first dimension; if an integer vertex delimits the polyhedron after adding this constraint, the scan will be completely avoided by a subsequent call.

i) Inclusion test

```
Bool LBLIncluded(A, B)
```

Check if A is included in B, if:
`LBLSimplifyEmpty(LBLDifference(A, B))` is empty.

j) Conversion to Z-domain

```
LBL *LBL2ZDomain(LBL *A)
```

Convert LBL A into a **Z**-domain, by building the union of **Z**-domains composing each single LBL of `A` with function `sLBL2ZDomain`, and normalizing the result.

6 Conclusion

We extended the works carried out over the years by Pugh, Le Verge, Quinton, Rajopadhye, Gautam, and Iooss, to implement robust and sound operations on arbitrary unions of LBLs in PolyLib. All those functions have been tested on a wide range of verified examples, using `valgrind` and a memory sanitizer to ensure consistent memory management.

Our implementation is both general and efficient. We took care to handle difficult problems internally while using the best possible version of each algorithm to uniformly manipulate all those objects: **Z**-polyhedra, **Z**-domains, single LBLs or their union. The implementation is available on the git repository of PolyLib [5].

Problems such as computing the cardinality, generating scanning loops, or finding the lexicographic minimum or maximum of a union of LBLs are not integrated in the library core functions. We are currently working on providing additional functions to the users to help them addressing these problems as efficiently as possible.

Acknowledgments

We are grateful to Patrice Quinton for insightful discussions and for being a key source of motivation in initiating this work.

We are grateful to Sven Verdoolaege for carefully rereading an earlier version of this paper and providing valuable comments that helped improve it.

References

[1] Gautam Gupta. *Some advances in the polyhedral model.* PhD thesis, Colorado State University, 2010.

[2] Gautam Gupta and Sanjay Rajopadhye. The Z-polyhedral model. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, page 237–248, New York, NY, USA, 2007. Association for Computing Machinery.

[3] Guillaume Iooss and Sanjay Rajopadhye. A library to manipulate Z-polyhedra in image representation. *Second International Workshop on Polyhedral Compilation Techniques (IMPACT)*, 2012.

[4] Hervé Le Verge. Recurrences on lattice polyhedra and their applications. Technical report, IRISA, 1995.

[5] Vincent Loechner. Polylib git: https://github.com/vincentloechner/polylib/.

[6] Vincent Loechner. PolyLib: A library for manipulating parameterized polyhedra. Technical report, Université de Strasbourg, 1999.

[7] Sunder Phani Kumar Nookala and Tanguy Risset. A library for Z-polyhedral operations. Technical Report. Rapport de Recherche Irisa, No1330, IRISA, 2000.

[8] William Pugh. The omega test: A fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91:Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pages 4–13, 1991.

[9] Patrice Quinton, Sanjay Rajopadhye, and Tanguy Risset. On manipulating Z-polyhedra using a canonical representation. *Parallel Processing Letters*, 07(02):181–194, 1997.

[10] Doran K. Wilde. A library for doing polyhedral operations. *Parallel Algorithms and Applications*, 15(3-4):137–166, 2000.