Basic Algorithms for Periodic-Linear Inequalities and Integer Polyhedra

Alain Ketterlin
Université de Strasbourg & Inria Grand-Est (France)

Abstract

This paper explores an alternative definition of linear inequalities over integer variables, focusing on expressive power and precision. The core technique is to use periodic numbers (also called Ehrhart numbers) to account for the seemingly irregular behavior of affine combinations of integer variables. The new representation is used to tighten inequalities, and from there derive a correct and complete decision procedure similar to Fourier-Motzkin elimination. A decomposition algorithm is also described: its result is a disjoint union of elementary polyhedra. Members of the union have a single contiguous range on each dimension, and are guaranteed to be non-empty. The data-structure used to represent the decomposition is similar to an abstract syntax tree. Several properties of this representation are briefly examined.

1 Introduction

The polyhedral model [9] has proved its ability to represent a large class of programs and their transformations. However, we have the (admittedly subjective) feeling that most of its fundamental algorithms do not fully account for an obvious characteristics of the polyhedra of interest in compilation: the fact that all involved variables are integers. Despite indisputable successes on that front, such as Pip [8] and Omega [14], we think this aspect needs further study.

This paper explores an alternative definition of linear inequalities over integer variables, slightly drifting away from linear programming techniques and focusing on expressive power and precision. Our starting point is the Fourier-Motzkin elimination algorithm, especially trying to avoid its incompleteness when applied to integer domains (Section 2). This will lead to a new representation of linear inequalities over integer variables, whose properties are examined in Section 3. A revised version of variable elimination is described in Section 4. Trying to derive a sensible projection algorithm leads to a decomposition algorithm, described in Section 5. The resulting decomposition directly provides various projections, as well as lexicographic extrema. Interestingly, this algorithm manipulates syntactic structures, and is almost entirely based on a single operation called *affine unswitching*.

IMPACT 2018 (http://impact.gforge.inria.fr/impact2018)
Eighth International Workshop on Polyhedral Compilation Techniques,
In conjunction with HiPEAC 2018.
January 23, 2018, Manchester, United Kingdom

The research described here is preliminary, and its only goal is the exploration of alternative representations and algorithms. We have developed software tools to validate core ideas, but none of these tools is mature enough to support meaningful comparison with existing machinery. Moreover, certain algorithms are obviously inapplicable beyond simple cases: we try to summarize the main limitations, and offer some directions for improvement, in the conclusion.

2 Motivation & Background

Our study starts with the Fourier-Motzkin variable elimination algorithm. Given a set $\{x_1, \ldots, x_n\}$ of variables with values in $\mathbb{Q}$, and a set of N linear inequalities

$$\left\{ c_{k0} + \sum_{i=1}^{n} c_{ki} x_i \geq 0 \,\middle|\, 1 \leq k \leq N \right\}$$

Fourier-Motzkin elimination proceeds by eliminating one variable after the other, until either producing a trivially false inequality (in which case the system is unsatisfiable), or running out of variables (in which case the system is guaranteed to have solutions). A single elimination step, targeting variable x_n, consists in:

1. for any pair of one lower and one upper bound
$$f_l(x_1, \ldots, x_{n-1}) \leq a x_n \quad b x_n \leq f_u(x_1, \ldots, x_{n-1}) \quad (1)$$
with $a, b > 0$, add the new *combined* inequality
$$b \cdot f_l(x_1, \ldots, x_{n-1}) \leq a \cdot f_u(x_1, \ldots, x_{n-1}) \quad (2)$$
2. remove all inequalities involving x_n

Every elimination step removes one variable. A trivially false inequality appearing during any elimination step is a necessary and sufficient condition for the system to have no solution, because (1) and (2) are logically *equivalent*.

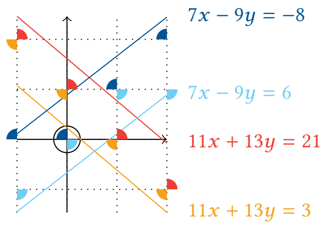
This algorithm is interesting for several reasons. First, it provides a simple decision procedure, proving or disproving the existence of solutions. Second, it also performs *projection*: what remains after eliminating variables $\{x_{p+1}, \ldots, x_n\}$ is a projection of the original polyhedron over $\{x_1, \ldots, x_p\}$. Third, the bounds that are combined (and then dropped) during an elimination step delimit the values of the eliminated variable in solutions. All these properties, however, are valid only when variables have rational values.

Adapting Fourier-Motzkin elimination to integer variables has given rise to the Omega test [14]. Variants of the same algorithm have been used to generate scanning code, which is a loop nest executing one statement for each integer point inside the polyhedron [2]. The same algorithm has even been extended to project sets of inequalities with parametric coefficients [11]. And it has probably many other uses.

We are going to use an example from the original Omega paper [14, Section 2.3.3] to try to understand the specifics of integer variables. The following polyhedron, called *Omega's nightmare* is used to illustrate a case where Omega needs three attempts before deciding it is empty:

$$\begin{array}{rcccl} 3 & \leq & 11x + 13y & \leq & 21 \\ -8 & \leq & 7x - 9y & \leq & 6 \end{array}$$

The graph is as follows:



Integer points for each constraint pair are depicted as small pie-slices. Note that the polyhedron does not contain any integer point, even though both of its (continuous) projections on the axes do.

The leftmost corner is the region delimited by constraints:

$$3 - 11x \leq 13y \quad \wedge \quad 9y \leq 7x + 8 \tag{3}$$

Combining these constraints leads to $-77 \leq 190x$, the exact (rational) x coordinate of this vertex. Rounding up this bound to the next integer leads $0 \leq x$, which is an under-estimation.

Omega correctly assesses emptiness by repeating the combination with refined inequalities, and finally enumerating a range of x coordinates which are tested individually. We will not discuss Omega further here, but rather focus on understanding what went wrong with straightforward variable elimination. Our analysis is as follows.

- Linear inequalities like $9y \leq 7x + 8$ are *loose* most of the times: the upper bound on $9y$ is precise only when $7x + 8$ is a multiple of 9, and otherwise leaves a "slack" of up to 8 units between $9y$ and its bound. Later combinations accumulate and amplify the slack.
- A consequence of the accumulation of slack is that the logical equivalence of a combination step is broken when dealing with integer variables. The missing implication (the only-if part) is:

  $$\begin{array}{l} b \cdot f_l(x_1, \ldots, x_{n-1}) \leq a \cdot f_u(x_1, \ldots, x_{n-1}) \\ \nrightarrow \exists x_n^*, \left\{ \begin{array}{l} f_l(x_1, \ldots, x_{n-1}) \leq a x_n^* \\ b x_n^* \leq f_u(x_1, \ldots, x_{n-1}) \end{array} \right. \end{array}$$

  This makes the classical Fourier-Motzkin test correct but incomplete on integers, failing to recognize some empty polyhedra.

Recognizing this problem also immediately offers a solution, at least in theory: tightening bounds. If one can assume tight bounds, that is if both bounds are (provably) multiples of the coefficient for all values of all other variables

$$\begin{array}{l} \exists f_l', \ f_l(x_1, \ldots, x_{n-1}) = a f_l'(x_1, \ldots, x_{n-1}) \\ \exists f_u', \ f_u(x_1, \ldots, x_{n-1}) = b f_u'(x_1, \ldots, x_{n-1}) \end{array}$$

then the existence of a value x_n^* such that

$$f_l'(\ldots) \leq x_n^* \leq f_u'(\ldots)$$

implies $b \cdot f_l(\ldots) \leq ab x_n^* \leq a \cdot f_u(\ldots)$, which implies the original bounds, and equivalence is restored.

Therefore, given two loose bounds

$$f_l(x_1, \ldots, x_{n-1}) \leq a x_n \qquad b x_n \leq f_u(x_1, \ldots, x_{n-1})$$

we can make Fourier-Motzkin elimination complete if we find a way to derive two equivalent tight inequalities, and combine these instead of the original ones. There are two "standard" ways to tighten inequalities:

- By using integer parts:

  $$\left\lceil \frac{f_l(x_1, \ldots, x_{n-1})}{a} \right\rceil \leq x_n \quad x_n \leq \left\lfloor \frac{f_u(x_1, \ldots, x_{n-1})}{b} \right\rfloor$$

  but the combination is difficult to manipulate, for instance to eliminate another variable.
- By using the euclidean remainder:

  $$\begin{array}{l} f_l'(\ldots) - \left(f_l'(\ldots) \bmod a\right) \leq a x_n \\ \qquad \text{where } f_l'(\ldots) = f_l(\ldots) + (a-1) \\ b x_n \leq f_u(\ldots) - (f_u(\ldots) \bmod b) \end{array}$$

  and then introduce one new variable and two inequalities *per* modulo expression, of which there is one per tightened equality, which is a lot.

Both solutions are equally awkward, and actually hopeless: we will see in Section 3.3 that combining two tightened inequalities does not always result in a single linear inequality. The next section describes a workable tightening mechanism for linear inequalities.

3 Periodic-Linear Inequalities

This section details a tightening process based on periodic numbers. It also explores the meaning and some properties of tightened inequalities.

3.1 Periodic Numbers and Expressions

A *periodic number* is a collection of integers indexed by the congruence class of an expression:

$$\langle v_0, v_1, \ldots, v_{\pi-1} \rangle_x^\pi = \left\{ \begin{array}{ll} v_0 & \text{if } x \equiv 0 \bmod \pi \\ v_1 & \text{if } x \equiv 1 \bmod \pi \\ \vdots & \\ v_{\pi-1} & \text{if } x \equiv (\pi - 1) \bmod \pi \end{array} \right.$$

where v_i are numbers (plain integers or periodic numbers), called the *elements*; π is a positive integer, called the *period* (or *size*); and x is any expression, usually an unknown, called the *argument*. Periodic numbers have been introduced by

Simple properties		
Linearity	$a + b \cdot \langle \dots, v_i, \dots \rangle_x^\pi = \langle \dots, a + b \cdot v_i, \dots \rangle_x^\pi$	
Rotation	$\langle v_0, v_1, \dots \rangle_{x+1}^\pi = \langle v_1, \dots, v_0 \rangle_x^\pi$	(with invariance: $\langle v_0, \dots \rangle_{ax+b}^\pi = \langle v_0, \dots \rangle_{(a \bmod \pi)x+(b \bmod \pi)}^\pi$)
Extension	$\langle v_0, \dots, v_{\pi-1} \rangle_x^\pi = \langle v_0, \dots, v_{\pi-1}, \dots, v_0, \dots, v_{\pi-1} \rangle_x^{c\pi}$	(with elements $v_0, \dots, v_{\pi-1}$ repeated c times)
Addition (etc.)	$\langle v_0, \dots \rangle_x^\alpha + \langle w_0, \dots \rangle_x^\beta = {}^i\left\langle \dots, v_{(i \bmod \alpha)} + w_{(i \bmod \beta)}, \dots \right\rangle_x^{\mathrm{lcm}(\alpha,\beta)}$	
Division	$\langle v_0, \dots \rangle_{cx}^\pi = {}^i\left\langle \dots, v_{(ci \bmod \pi)}, \dots \right\rangle_x^{\pi/\gcd(\pi,c)}$	(special case: $\langle v_0, \dots \rangle_{cx}^{c\pi} = {}^i\langle \dots, v_{ci}, \dots \rangle_x^\pi$)
Multidimensional properties		
Transposition	$\left\langle \dots, \left\langle \dots, v_{ij}, \dots \right\rangle_x^\alpha, \dots \right\rangle_y^\beta = \left\langle \dots, \left\langle \dots, v_{ji}, \dots \right\rangle_y^\beta, \dots \right\rangle_x^\alpha$	
Distribution	$\langle v_0, \dots, v_{\alpha-1} \rangle_{\langle w_0, \dots, w_{\beta-1} \rangle_x^\beta}^\alpha = \left\langle \langle v_0, \dots, v_{\alpha-1} \rangle_{w_0}^\alpha, \dots, \langle v_0, \dots, v_{\alpha-1} \rangle_{w_{\beta-1}}^\alpha \right\rangle_x^\beta$	
Separation	$\langle v_0, v_1, \dots, v_{\alpha-1} \rangle_{x+y}^\alpha = \left\langle \langle v_0, \dots, v_{\alpha-1} \rangle_y^\alpha, \langle v_1, \dots, v_0 \rangle_y^\alpha, \dots, \langle v_{\alpha-1}, \dots, v_{\alpha-2} \rangle_y^\alpha \right\rangle_x^\alpha$	

Table 1. Essential operations on periodic numbers: a, b, v_*, w_* are integers; c, α, β, π are positive integers; x and y are unknowns; ${}^i\langle \dots \rangle^\pi$ indicates that i denotes the position of the element (in $[0, \pi - 1]$) where it appears.

Ehrhart in its classic work on counting integer points inside a polytope [6, 7], and have been generalized by Clauss in the context of the polyhedral model [5]. We know of very little later work making use of periodic numbers. One exception is the work by Meister [13] on computing the convex hull of a polyhedron, which is a precursor of our approach.

Basic symbolic operations on periodic numbers are collected in Table 1. The rest of this paper relies on the ability to normalize the representation of periodic numbers which take symbolic expressions as argument. The *separation* rule in Table 1 is a cornerstone: along with *division* and *extension*, it states that when the argument is a linear function of several variables, a periodic number can be made multidimensional, with every dimension involving a single variable with unit coefficient. For instance, the periodic number:

$$\langle 0, 1, 2 \rangle_{2x+6y+5z-1}^3$$

can be rewritten as:

$$\left\langle \left\langle \langle 2, 1, 0 \rangle_x^3 \right\rangle_y^1, \left\langle \langle 1, 0, 2 \rangle_x^3 \right\rangle_y^1, \left\langle \langle 0, 2, 1 \rangle_x^3 \right\rangle_y^1 \right\rangle_z^3$$

In the following, the superscript indicating period will often be omitted. Likewise, periodic numbers with period 1 will be replaced with their unique component. They appear here because $6y$ is a multiple of 3, and have been left in to illustrate the nesting along the list of unknowns.

We are going to manipulate symbolic expressions over a set of unknowns, typically parameters and variables, listed in an arbitrary but fixed order. Given such an ordered list $[x_1, \dots, x_n]$, a *periodic-linear expression* (PLE) in *normal form* is an expression of the form

$$\left(\sum_{i=1}^{n} a_i x_i \right) + \left\langle \cdots, \left\langle \cdots \langle \cdots \rangle_{x_1}^{\pi_1} \cdots \right\rangle_{x_{n-1}}^{\pi_{n-1}}, \cdots \right\rangle_{x_n}^{\pi_n} \qquad (4)$$

that is, an affine expression where the constant has been replaced with an n-dimensional periodic number with unknowns as arguments, conventionally nested in reverse order. "Usual" affine expressions are PLEs where $\pi_1 = \ldots = \pi_n = 1$.

It is often convenient to use the *simplified normal form*:

$$a_n x_n + \left\langle \cdots, a_{n-1} x_{n-1} + \langle \cdots \rangle_{x_{n-1}}^{\pi_{n-1}}, \cdots \right\rangle_{x_n}^{\pi_n} \qquad (5)$$

where every *linear component* $a_i x_i$ is placed near its corresponding *periodic component*. This form is redundant (all occurrences of $a_i x_i$ are identical) but much simpler and abstract when it comes to manipulate expressions with respect to x_n only, because the elements of a periodic component are all invariant with respect to the argument.

Addition and subtraction between PLEs, as well as multiplication by an integer, are all stable operations: the result is a PLE. Less obvious is the fact that, given integers $v_0, \dots, v_{\pi-1}$ and a PLE X, the expression $\langle v_0, \dots, v_{\pi-1} \rangle_X^\pi$ is also a PLE: its normal form is obtained by the recursive application of a succession of separation, extension, division, and distribution operations. The details of the derivation are omitted, they are purely technical. Another useful stability property is that given two PLEs X and Y over the same set of variables, replacing variable x_i by Y in X is guaranteed to be a PLE.

Finally, we are going to build periodic-linear inequalities by comparing a PLE with 0. The inequality $a x_n + \langle \dots \rangle_{x_n}^\pi \geq 0$ in simplified normal form will be called:

- *linear* if $a \neq 0$ and $\pi = 1$;
- (purely) *periodic* if $a = 0$ and $\pi > 1$;
- *mixed* if $a \neq 0$ and $\pi > 1$;

Other inequalities, not involving x_n, will be called *uniform*. Since these qualifiers are common words, we keep them italicized throughout the paper when they refer to the form of a periodic-linear inequality.

3.2 Tightening

As per the definition of periodic numbers, the following holds for any given PLE X:

$$\langle 0, 1, \ldots, \pi - 1\rangle_X^\pi = X \bmod \pi$$

Hence, the largest multiple of π less than or equal to X is:

$$X - \langle 0, 1, \ldots, \pi - 1\rangle_X^\pi$$

and the smallest multiple of π greater than or equal to X is:

$$\begin{aligned} & (X + \pi - 1) - \langle 0, 1, \ldots, \pi - 1\rangle_{X+\pi-1}^\pi \\ = \quad & X + \langle 0, \pi - 1, \ldots, 1\rangle_X^\pi \end{aligned}$$

This simple remark provides a tightening strategy for *linear* inequalities: if $a > 0$ and X is a PLE over $[x_1, \ldots, x_{n-1}]$, then:

$$ax_n \le X \leftrightarrow ax_n \le X - \langle 0, 1, \ldots, a - 1\rangle_X^a \quad (6)$$

$$X \le ax_n \leftrightarrow X + \langle 0, a - 1, \ldots, 1\rangle_X^a \le ax_n \quad (7)$$

The proof of these equivalences is immediate, by reasoning by case on the congruence of X modulo a.

For instance, Omega's nightmare's left corner inequalities (see Equation (3) in Section 2) can be tightened by adding a simple corrective term:

$$\begin{aligned} 3 - 11x \le 13y \quad &\Rightarrow \quad 3 - 11x + \langle 0, 12, \ldots, 1\rangle_{3-11x}^{13} \le 13y \\ 9y \le 7x + 8 \quad &\Rightarrow \quad 9y \le 7x + 8 - \langle 0, 1, \ldots, 8\rangle_{7x+8}^{9} \end{aligned}$$

These tight inequalities can be safely combined without introducing any slack:

$$\begin{gathered} 9 \cdot (3 - 11x + \langle 0, 12, \ldots, 1\rangle_{3-11x}^{13}) \\ \le 9 \cdot 13 \cdot y \le \\ 13 \cdot (7x + 8 - \langle 0, 1, \ldots, 8\rangle_{7x+8}^{9}) \end{gathered}$$

It takes some algebraic properties of periodic numbers to rearrange this inequality. Please admit the following result:

$$\langle 117, 73, 29, -15, -59, 14, \ldots, 59, 15, -29, 44\rangle_x^{117} \le 190x \quad (8)$$

which is a *mixed* inequality with respect to x: there is a different bound for each congruence of x modulo 117.

We now come to the trickiest part of tightening. Equation (8) is a combination result, and should be tightened before being further combined. Consider phase 2 for instance, where $29 \le 190x$. Straightforward tightening results in

$$1 \le x$$

but this is again loose, because it ignores the phase: this bound is not suitable for x congruent to 2 modulo 117. Taking the phase into account means that $x = 117x' + 2$ for some x'. Performing this change of variable leads to $29 \le 190(117x' + 2)$, and tightening with respect to x' leads $0 \le x'$. Reverting the change of variable leads to the correct result:

$$2 \le x$$

A similar change of variable must be done for all 117 phases.

The lesson learned with this example is that tightening a *mixed* periodic-linear inequality is slightly more complex than simple *linear* tightening. Fortunately, it has a closed form, also covering the linear case. Given an upper bound

$$a_n x_n \le \langle A_0, \ldots\rangle_{x_n}^{\pi_n} \quad (9)$$

where $A_0, A_1, \ldots$ are all PLEs over variables $[x_1, \ldots, x_{n-1}]$, the result of tightening is the following inequality:

$$a_n x_n \le {}^i\left\langle \ldots, A_i - \langle 0, 1, \ldots, a_n\pi_n - 1\rangle_{A_i - ia_n}^{a_n\pi_n}, \ldots\right\rangle_{x_n}^{\pi_n} \quad (10)$$

where i, ranging from 0 to $\pi_n - 1$, designates the phase. There is a similar formula for lower bounds, which we omit. The proof that inequalities (9) and (10) are equivalent is by case on the congruence of x_n modulo π_n: if $x_n = \pi_n x'_n + \varphi$, then the equivalence becomes:

$$\begin{aligned} & a_n(\pi_n x'_n + \varphi) \le A_\varphi \leftrightarrow \\ & \qquad a_n(\pi_n x'_n + \varphi) \le A_\varphi - \langle 0, 1, \ldots, a_n\pi_n - 1\rangle_{A_\varphi - \varphi a_n}^{a_n\pi_n} \end{aligned}$$

which, after some rearrangement, is exactly covered by *linear* tightening equivalence, in Equation (6) above.

Coming back to our example, Equation (8) is tightened as

$$\langle 117, 1, 2, 3, 4, \ldots, 112, 113, 114, 115, 116\rangle_x^{117} \le x \quad (11)$$

Intuition suggests that this actually means $1 \le x$; Section 3.4 explains how this simpler inequality can be automatically inferred. In the meantime, note that tightening alone is enough to correct the under-estimation observed during the rational-with-rounding application of Fourier-Motzkin elimination in Section 2. For lack of space, the reader will have to admit that combining inequalities forming the "right corner" leads to $x \le 0$ with similar means, from which Omega's nightmare emptiness follows.

3.3 Saturating Inequalities

Before going back to adapt Fourier-Motzkin to integer polyhedra, let us examine a few characteristics of tightened inequalities. In this and later section we use a second example polyhedron, named LS(N):

$$\begin{array}{lcccccl} (A) & x + 2 & \le & 3y & \le & x + 5 & (C) \\ (B) & x - N + 1 & \le & 2y & \le & x + 1 & (D) \end{array}$$

with upper bounds on y tightened as:

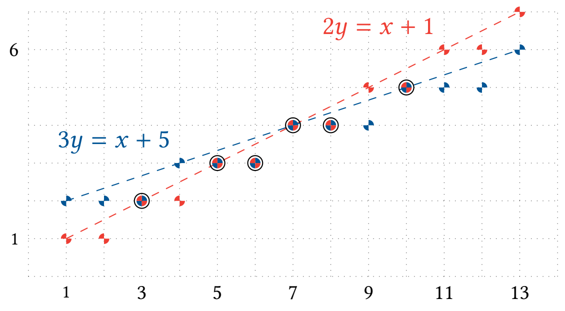
$$\begin{array}{lcll} 3y & \le & x + 5 - \langle 2, 0, 1\rangle_x & (C') \\ 2y & \le & x + 1 - \langle 1, 0\rangle_x & (D') \end{array}$$

We say that a bound is *saturated* when the inequality is actually an equality. In the case of upper bounds, the saturated bound gives, for each x, the maximal value of y for which the inequality holds.

We define a π-vertex (of a 2D-polyhedron) as the lieu where two bounds are saturated simultaneously. For bounds (C') and (D'), we can write the equality and solve for x:

$$\begin{aligned} & 2(x + 5 - \langle 2, 0, 1\rangle_x) = 3(x + 1 - \langle 1, 0\rangle_x) \\ \Rightarrow \quad & x = \langle 6, 7, 8, 3, 10, 5\rangle_x \end{aligned}$$

The resulting equation involves x both as a plain variable *and* as the argument of a periodic number. The definition of periodic numbers gives a precise meaning to such equations: $x = 6$ when $x \equiv 0 \bmod 6$, $x = 7$ when $x \equiv 1 \bmod 6$, and so on. Here, all phases are valid solutions, but this is not necessarily the case in general. The geometric meaning of this equation appears on the following graph:



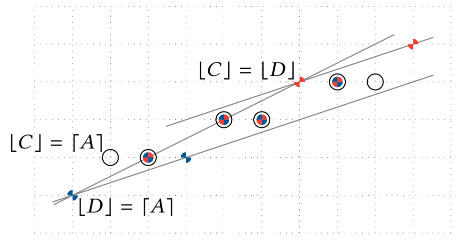
The butterfly-like markers designate integer points that saturate one inequality, and full circles designate points where both inequalities are saturated (the π-vertex).

An important remark about this example is that the π-vertex is made of six distinct integer points, and that these points are not connected. A consequence of this is that there is no single *integer* x^* such that one bound "dominates" for $x < x^*$ and the other dominates for $x^* \leq x$: partitioning *is* possible, but the boundary is a periodic number. Finally, note that we have tightened *with respect to* y and solved for x: had we considered the variables in the reverse order, the π-vertex would have been formed of a single point ($y = 4$).

Another example of the complexity of intersecting inequalities is the case of bound (A), tightened as

$$x + 2 + \langle 1, 0, 2\rangle_x \leq 3y$$

and intersected with (D'). The resulting π-vertex is found at $x = \langle 6, 1, 8, 3, 4, 5\rangle_x$. This π-vertex intersects with the one pictured above, at $x = 3, 5, 6, 8$, which shows that several π-vertices can share integer points.



$\lfloor C\rfloor = \lfloor D\rfloor$
$\lfloor C\rfloor = \lceil A\rceil$
$\lfloor D\rfloor = \lceil A\rceil$

Finally, consider the π-vertex formed by saturating both (A) and (C), displayed as circles above, whose equation is:

$$\langle 3, 0, 3\rangle_x = 3 \qquad \text{(which means } x \not\equiv 1 \bmod 3)$$

On integer domains, parallel (and distinct) saturated inequalities can have a non-empty, and even infinite intersection.

The purpose of this little exercise was to convince the reader that nothing is simple when dealing with integers, and that intuitions forged by looking at rational polyhedra hardly transpose to the integer case.

3.4 Mixed Inequalities and Disjunction

This section looks at one last important property of tightened inequalities, especially of the *mixed* category: their tendency to hide disjunctions. For instance, consider the result of combining first (A) and (D), and then (B) and (C) in the example of the previous section. The result is:

$$(AD) \quad \langle 6, 1, 8, 3, 4, 5\rangle_x \leq x$$
$$(BC) \quad x \leq 3N + \left\langle \begin{matrix} \langle 0,3\rangle_N, \langle 7,4\rangle_N, \langle 2,5\rangle_N, \\ \langle 3,0\rangle_N, \langle 4,7\rangle_N, \langle 5,2\rangle_N \end{matrix} \right\rangle_x^6$$

Looking at inequality (AD), intuition tells that it is equivalent to $(x = 1) \vee (3 \leq x)$. Inequality (BC) is more complex, but turns out to mean $(x \leq 3N + 5) \vee (x = 3N + 7)$.

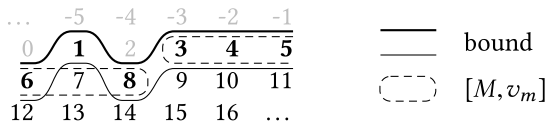
The algorithm to systematically convert *mixed* inequalities into disjunctions of *linear* inequalities is called Disjoin. It is defined inductively on the dimension of the periodic number. In the one-dimensional case, assuming the inequality is a lower-bound of the form

$$\langle v_0, \ldots\rangle_x^\pi \leq ax \qquad \text{with } v_i\text{s plain integers}$$

the algorithm proceeds as follows:

```
Disjoin_1($\langle v_0, \ldots\rangle_x^\pi \leq ax$)
    let $M = v_m - a(\pi - 1)$, where $v_m = \max\{v_i\}$
    let $O = \{v_i \mid v_i < M\}$
    return $(M \leq ax) \vee \left(\bigvee_{d \in O} (x = d)\right)$
```

Value v_m is the maximal value appearing in the periodic number: it is equal to 8 for (AD). Value M is the start of a full period (of length π) ending at v_m: the interval $[3, 8]$ for (AD). The set O collects "outliers", values which are outside of the period anchored at M and need special casing: it is the set $\{1\}$ for (AD). Here is a pictured summary, with columns representing congruence classes modulo 6, illustrating the fact that values above $M = 3$ are necessarily covered:



The final result is a lower bound on M, plus a set of special equality relations for outliers (which can sometimes be turned into short intervals). Inequality (AD) is equivalent to $(x = 1) \vee (3 \leq x)$.

Another example is Omega's nightmare's lower bound on x (see Equation (11) in Section 3.2). This bound has $M = 1$ and no outlier: it is therefore turned into $1 \leq x$.

A full description of the multidimensional case would require heavy notations. We are only going to illustrate it on the (BC) inequality above:

1. Transpose the periodic number to bring the target variable at the deepest level, and consider the inequality to be local to this level. This turns (BC) into:

$$3N + \langle\, x \leq \langle 0,7,2,3,4,5\rangle_x \,,\; x \leq \langle 3,4,5,0,7,2\rangle_x \,\rangle_N$$

This strange notation highlights the fact that, after transposition, the next-to-last level of the periodic number is a collection of one-dimensional inequalities on x.

2. Apply the one-dimensional algorithm to each inner inequality, leading to:

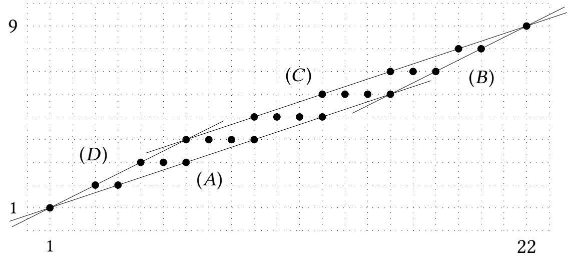
$$3N + \langle \quad (x \leq 5) \vee (x = 7), \quad (x \leq 5) \vee (x = 7) \quad \rangle_N$$

3. Transpose back into the original order:

$$(x \leq 3N + \langle 5,5\rangle_N) \vee (x = 3N + \langle 7,7\rangle_N)$$

Note that this transposition phase also needs to simplify redundant periodic numbers along the way.

Inequalities (AD) and (BC) are the results of eliminating variable y. The fact that they are disjunctions means that LS(N) has holes along the x axis. This can be verified by plotting the integer points for some value of N (5 here):



As expected, there is no integer point for $x = 2$ and $x = 21$.

This section contains only sketchy algorithm descriptions, but the main ideas are fairly simple. The point was to show that *mixed* inequalities can always be transformed into a disjunction of *linear* inequalities (or equalities in many cases). *Mixed* inequalities are therefore "almost" linear, in the sense that they display vagaries for a finite number of points only. They are really specific to integer domains, which lack the inherent smoothness of their rational (or real) counterparts.

Here is one last example, taken from the Omega paper [14, Section 4]: eliminate b in

$$5b \leq a \leq 6b$$

The proposed solution is:

$$\{20 \leq a\} \vee \{0 \leq a; a = 6\alpha\} \vee \{1 \leq a; a = 6\alpha + 1\}$$
$$\vee \{2 \leq a; a = 6\alpha + 2\} \vee \{3 \leq a; a = 6\alpha + 3\}$$

which is a disjunction based on the congruence classes of a modulo 6, with α an existential variable ranging over integers. In contrast, periodic-linear tightening leads to:

$$a + \langle 0,5,4,3,2,1\rangle_a \leq 6b \qquad 5b \leq a - \langle 0,1,2,3,4\rangle_a$$

which combines into:

$$\langle\, \underline{0}, 31,32,33,34, \underline{5,6}, 37,38,39, \underline{10,11,12}, 43,44,$$
$$\underline{15,16,17,18}, 49, \underline{20,21,22,23,24,25,26,27,28,29} \,\rangle_a^{30} \leq a$$

with outliers underlined. Our implementation of Disjoin groups outliers into intervals when possible, producing:

$$(a = 0) \vee (5 \leq a \leq 6) \vee (10 \leq a \leq 12) \vee (15 \leq a \leq 18) \vee (20 \leq a)$$

Note that no existential variable is needed, and that the result is a disjoint union.

4 The Omicron Test

We are now ready to adapt Fourier-Motzkin elimination to integer domains. We have called the resulting procedure the Omicron test. There are two major changes in the basic algorithm. First, tightening is applied systematically, on input inequalities as well as on all inequalities created during the process. Second, inequalities with a non-trivial periodic component are turned into disjunctions, which lead to *forking* the system into several, independent sub-systems that are tested individually, maybe in parallel. This forking process is similar to what Omega does after two inconclusive attempts to solve the original system.

Non strictly *linear* inequalities need special treatment. Over variables $[x_1, \ldots, x_n]$, assuming the algorithm is about to eliminate x_n, Omicron does the following, in order:

- If the system contains a *periodic* inequality

$$0x_n + \langle X_0, \ldots\rangle_{x_n}^{\pi_n} \geq 0 \qquad \text{with } \pi_n > 1$$

then create π_n new systems where x_n is changed into $\pi_n x'_n + \varphi$, with $\varphi = 0, 1, \ldots, \pi_n - 1$. Every change of variable turns the *periodic* inequality into a *uniform* inequality $X_\varphi \geq 0$. This is called *splintering*, and essentially amounts to reason by case on the various phases of x_n modulo π_n.

- If the system contains a *mixed* inequality

$$a_n x_n + \langle X_0, \ldots\rangle_{x_n}^{\pi_n} \geq 0 \qquad \text{with } a_n \neq 0, \pi_n > 1$$

then apply the Disjoin algorithm (Section 3.4), and create one new sub-system for each term of the resulting disjunction. This is called *disjoining*, and essentially amounts to split the system around the holes, if any.

When only *linear* inequalities (lower and upper bounds) remain, an elimination step is performed, exactly as in the original Fourier-Motzkin algorithm.

Figure 1 shows two possible executions of Omicron on Omega's nightmare. The one on the left ignores Disjoin, and applies splintering on both period-117 inequalities, leading to 117 very simple systems. This shows that splintering can also be applied to *mixed* inequalities, and can replace disjoining, at the cost of additional forking. The right part of Figure 1 shows that Disjoin turns the *mixed* inequalities into simple bounds (there are no outliers after tightening), and a single elimination step concludes on emptiness.

Figure 2 is Omicron applied to LS(N). Splintering happens early on x, because of the *periodic* inequality

$$\langle\langle 2,1\rangle_N, \langle 0,1\rangle_N\rangle_x \leq N$$

Later elimination steps proceed smoothly, with both subsystems testing positively for some values of N. The two legs represent different parities of x: the left leg (x even) has no solution for $N = 0$.

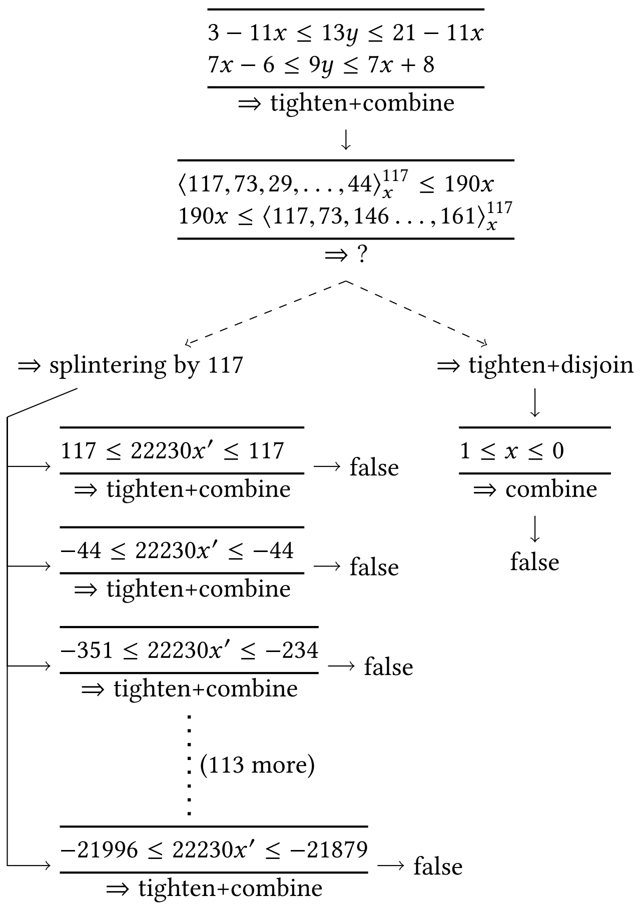

$3 - 11x \le 13y \le 21 - 11x$
$7x - 6 \le 9y \le 7x + 8$
⇒ tighten+combine
$\langle 117, 73, 29, \dots, 44\rangle_x^{117} \le 190x$
$190x \le \langle 117, 73, 146 \dots, 161\rangle_x^{117}$
⇒ ?
⇒ splintering by 117
⇒ tighten+disjoin
$117 \le 22230x' \le 117$
⇒ tighten+combine
false
$1 \le x \le 0$
⇒ combine
false
$-44 \le 22230x' \le -44$
⇒ tighten+combine
false
$-351 \le 22230x' \le -234$
⇒ tighten+combine
false
(113 more)
$-21996 \le 22230x' \le -21879$
⇒ tighten+combine
false

Figure 1. Two different decisions of Omega's nightmare.

5 Decomposition

By its very nature, variable elimination seems to be a sensible mean to project a polyhedron onto a subset of its dimensions. However, it is not actually the case, because OMICRON may fork a system into (disjoint) sub-systems, where most of the inequalities are replicated. This appears clearly in Figure 2, where the next-to-last tree level (after the elimination of both y and x) has two largely overlapping systems. Moreover, mere projection hides some details about the polyhedron. For instance, the LS(N) polyhedron changes its effective bounds on y several times along the x axis: see the graph in Section 3.3. Capturing such changes allows splitting the original polyhedron into elementary fragments, where each dimension has a single contiguous range. Projection can be recovered from such a decomposition, by merging ranges where discrimination is superfluous.

5.1 Polyhedra and Abstract Syntax Trees

A decomposition needs to maintain a set of polyhedra, organized inside a progressive subdivision on ranges of the variables, considered in a predefined order. An adequate data

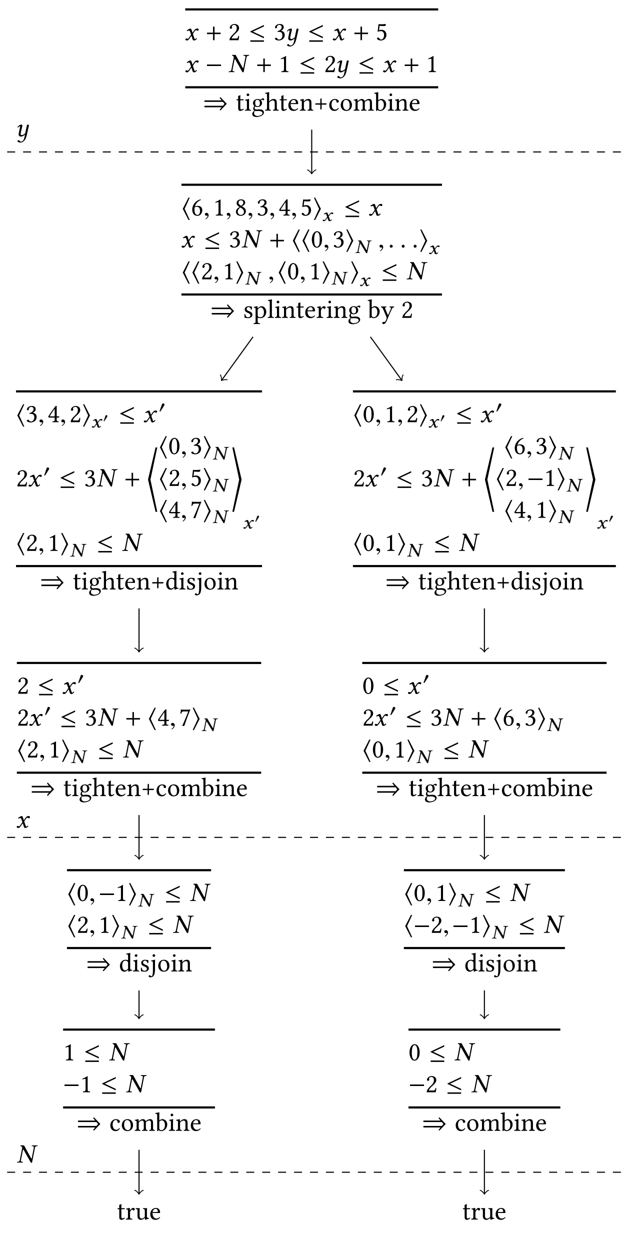

$x + 2 \le 3y \le x + 5$
$x - N + 1 \le 2y \le x + 1$
⇒ tighten+combine
y
$\langle 6, 1, 8, 3, 4, 5\rangle_x \le x$
$x \le 3N + \langle\langle 0, 3\rangle_N, \dots\rangle_x$
$\langle\langle 2, 1\rangle_N, \langle 0, 1\rangle_N\rangle_x \le N$
⇒ splintering by 2
$\langle 3, 4, 2\rangle_{x'} \le x'$
$2x' \le 3N + \left\langle \langle 0,3\rangle_N \; \langle 2,5\rangle_N \; \langle 4,7\rangle_N \right\rangle_{x'}$
$\langle 2, 1\rangle_N \le N$
⇒ tighten+disjoin
$\langle 0, 1, 2\rangle_{x'} \le x'$
$2x' \le 3N + \left\langle \langle 6,3\rangle_N \; \langle 2,-1\rangle_N \; \langle 4,1\rangle_N \right\rangle_{x'}$
$\langle 0, 1\rangle_N \le N$
⇒ tighten+disjoin
$2 \le x'$
$2x' \le 3N + \langle 4, 7\rangle_N$
$\langle 2, 1\rangle_N \le N$
⇒ tighten+combine
$0 \le x'$
$2x' \le 3N + \langle 6, 3\rangle_N$
$\langle 0, 1\rangle_N \le N$
⇒ tighten+combine
x
$\langle 0, -1\rangle_N \le N$
$\langle 2, 1\rangle_N \le N$
⇒ disjoin
$\langle 0, 1\rangle_N \le N$
$\langle -2, -1\rangle_N \le N$
⇒ disjoin
$1 \le N$
$-1 \le N$
⇒ combine
$0 \le N$
$-2 \le N$
⇒ combine
N
true
true

Figure 2. OMICRON on LS(N).

structure for the task is an abstract syntax tree (AST) formed of loop-like and if-then-else-like constructs, along with sequential composition of constructs. The AST is layered, with one layer per variable, and every one of its leaves represents one fragment of the decomposition. For example, a concrete initial representation of the LS(N) polyhedron, defined over parameter N and regular variables x and y, would be:

```
when _ <= N <= _
    for _ <= x <= _
        for _ <= y <= _
            if (...) then
                exec S(x,y)
```

The `(...)` condition attached to the `if`-construct is meant to contain the various inequalities defining the polyhedron. A `when` construct represents a range for a parameter, and `for` the range of a variable. The `_` symbol represents a missing bound, and can be understood as positive of negative infinity.

Variable ranges (`for` and `when`) carry a variable name, a *scale*, and two bounds, which are periodic-linear expressions over the set of enclosing variables. For instance

```
for 2x + [x:6,4,8] <= 6y <= 3x + [x:0,3]
```

represents a consecutive range of integers for `y`. In particular, the scale 6 is *not* a step: it is there only to ensure that both bounds can be kept tight at all times.

Conditions on `if` statements are arbitrary logical combinations of inequalities. All inequalities are kept tight at all times (as do range bounds). Additionally, a *mixed* inequality is turned into a disjunction (via Disjoin) at creation time, so that all inequalities in the AST are either *linear* or *periodic*.

5.2 Affine Unswitching

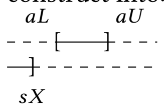
The polyhedron decomposition algorithm is a variation of Fourier-Motzkin elimination, with two differences. The first (minor) difference is that the algorithm "eliminates" inequalities rather than variables. The second (major) difference, is that it does not only combine lower bounds with upper bounds, but also "compares" two lower (or two upper) bounds to discriminate ranges of variables depending on which bound is effective. Both operations are performed at once by *affine unswitching*, which is a way to *hoist* an inequality out of its nearest enclosing range, or, more precisely, out of the nearest enclosing range of a variable or parameter appearing in the inequality.

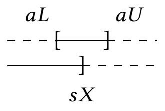
As we have mentioned earlier, all inequalities in the AST are either *linear* or *periodic*. Affine unswitching applies to *linear* inequalities. Consider such an inequality, an upper bound here, and its nearest enclosing range construct:

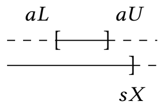
```
    for/when   $L \le sx_n \le U$   do
        ...    $ax_n \le X$    ...
```

where L, U, and X are arbitrary expressions not involving x_n. Affine unswitching transforms this construct into:

```
if $sX < aL$ then
    for $L \le sx_n \le U$ do
        ... $\boxed{ax_n \le X}$ ... // = false
else if $sX < aU$ then
    for $aL \le asx_n \le sX$ do
        ... $\boxed{ax_n \le X}$ ... // = true
    for $s(X + a) \le asx_n \le aU$ do
        ... $\boxed{ax_n \le X}$ ... // = false
else // $sX \ge aU$
    for $L \le sx_n \le U$ do
        ... $\boxed{ax_n \le X}$ ... // = true
```







There is an exactly symmetric definition of affine unswitching dedicated to lower bounds, which we omit.

Essentially, this transformed code covers all possible orderings of the bound provided by the inequality (X) and the range bounds (L and U), and determines all possible ranges where the inequality has an uniform truth value. On these ranges, the inequality can be replaced with a boolean constant. New guards are placed above variations of the original construct: by construction, they do not involve x_n, and ensure that the new ranges are non-empty. Condition $sX < aL$ is (a slight variation on) the result of a Fourier-Motzkin combination step, while $sX < aU$ is discriminating between two competing upper bounds, breaking the range of x_n to ensure a uniform truth value of the original inequality.

Affine unswitching applies to *linear* inequalities, but actually also to *periodic* inequalities, such as:

$$\langle X_0, X_1, \ldots \rangle_{x_n}^{\pi_n} \ge 0$$

which can be viewed as a collection of inequalities:

$$\langle X_0 \ge 0,\ X_1 \ge 0,\ \ldots \rangle_{x_n}^{\pi_n}$$

These inner inequalities $X_0 \ge 0$, $X_1 \ge 0$, ..., are hoisted first, eventually turning the original *periodic* inequality into a "periodic boolean" $\langle b_0, b_1, \ldots \rangle_{x_n}$, where $b_i \in \{\texttt{true}, \texttt{false}\}$. At this point, unless x_n is a parameter, the enclosing range on x_n is unrolled by a factor of π_n, and the condition finally vanishes. We omit the details. Note that unrolling applies to regular variable ranges only: for parameters, the periodic boolean is kept as is. This is the only difference between `when` and `for`.

It is important to realize that affine unswitching (and unrolling) applies without restriction. Any inequality can be eventually hoisted from its nearest enclosing range. The decomposition algorithm simply hoists inequalities repeatedly. All inequalities bubble up the AST, losing at least one variable during every unswitching, and eventually evaporate through the root. What remains is an AST that is totally free of conditionals (not even min / max in range bounds), and where all ranges are guaranteed to contain at least one integer value: this is the decomposition of the original polyhedron.

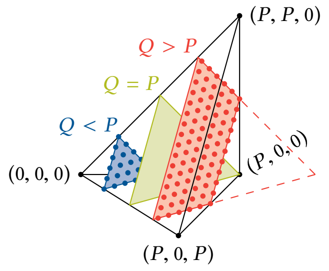
The decomposition of LS(N) is shown in Figure 3 (for reference, a plot of LS(5) appears in Section 3.4). Single-value ranges have been elided to save space : their unique value appears in the `exec` statement. The striking characteristic of this code is its extreme fragmentation: it comes from the stormy intersections between the two upper bounds on y, and symmetrically between the lower bounds. The smallest N for which a significant range appears in between these two intersections is $N = 5$. For the same reason, all values of N between 1 and 4 need to be special-cased. When $N \ge 5$, the stable body of this polyhedron is with x between 11 and $3N - 3$. Across this range, the range on y has width $((x + \langle 3,5,4 \rangle_x) - (x + \langle 3,2,4 \rangle_x))/3 + 1 = \langle 1,2,1 \rangle_x$. Dividing by the scale is necessarily exact since bounds are tight.

```
when N = 0
  exec S(1,1); exec S(3,2); exec S(5,3); exec S(7,4)
when N = 1
  exec S(1,1)
  for 3 <= x <= 8
      exec S(x,x+[x:0,1]//2)
  exec S(10,5)
when N = 2 [...]
when N = 3 [...]
when N = 4 [...]
when 5 <= N <= _
  exec S(1,1)
  for 3 <= x <= 8
      for 2x+[x:6,4,8] <= 6y <= 3x+[x:0,3]
          exec S(x,y)
  exec S(9,4)
  for 4 <= y <= 5
      exec S(10,y)
  for 11 <= x <= 3N-3
      for x+[x:3,2,4] <= 3y <= x+[x:3,5,4]
          exec S(x,y)
  for N <= y <= N+1
      exec S(3N-2,y)
  exec S(3N-1,N+1)
  for 3N <= x <= 3N+5
      for 3x-3N+[x:[N:6,3],[N:3,6]] <= 6y <= 2x+[x:6,10,8]
          exec S(x,y)
  exec S(3N+7,N+4)
```

Figure 3. Decomposition of LS(N). Indentation is nesting.

Another example is the following polyhedron, with parameters P and Q, the intersection of a pyramid and a plane:

$$\begin{cases} 0 \le i \le P \\ 0 \le j \le i \\ 0 \le k \le i - j \\ Q = i + j + k \end{cases}$$



Repeated affine unswitching produces:

```
when P = 0
  when Q = 0
      exec S(0,0,0)
when 1 <= P <= _
  when 0 <= Q <= P
      for Q+[Q:0,1] <= 2i <= 2Q
          for 0 <= j <= -i+Q
              exec S(i,j,-j-i+Q)
  when P+1 <= Q <= 2P
      for Q+[Q:0,1] <= 2i <= 2P
          for 0 <= j <= -i+Q
              exec S(i,j,-j-i+Q)
```

Decomposition happens in parameter space. When $P \ge 1$, the algorithm correctly discriminates between "triangular" ($0 \le Q \le P$) and "quadrilateral" ($P + 1 \le Q \le 2P$) regions.

Our last example illustrates periodic booleans and unrolling. It starts from a union, implemented as:

```
for _ <= t <= _
    for _ <= i <= _
        if t = 2i then
            exec S1(t,i)
        if t = 2i+1 then
            exec S2(t,i)
```

In our implementation, equality relations are treated as pairs of inequalities. Note that both t and i have unbounded ranges. After some affine unswitching, the AST becomes:

```
for _ <= t <= _
    if [t:true,false] then
        exec S1(t)(t+[t:0,-1]//2)
    if [t:false,true] then
        exec S2(t)(t+[t:0,-1]//2)
```

The next step is to unroll the loop over t, leading to:

```
for _ <= 2t <= _
    exec S1(2t)(t)
    exec S2(2t+1)(t)
```

5.3 Applications

A decomposition actually provides a projection, at the cost of some fragmentation (again). In Figure 3, the projection onto $[N, x]$ is easily computed by pruning the AST below ranges on x. What remains is the set $\{1, 3, 5, 7\}$ for $N = 0$, the union $\{1\} \cup [3, 8] \cup \{10\}$ for $N = 1$, and

$$\{1\} \cup [3, 8] \cup \{9\} \cup \{10\} \cup [11, 3N - 3] \cup \\ \{3N - 2\} \cup \{3N - 1\} \cup [3N, 3N + 5] \cup \{3N + 7\}$$

for $N \ge 5$, which is a ridiculous but accurate way to represent $\{1\} \cup [3, 3N + 5] \cup \{3N + 7\}$. Fragmentation is unavoidable in the general case though, because the projection of a disjoint union is not necessarily a disjoint union. Moreover, it is very easy to "stitch" together the various pieces, but a fully general description of the algorithm would take us too far.

More interesting is the fact that decomposition provides lexical minimum and maximum with no further processing. Inside each parameter domain, keeping the lower bound (resp. upper bound) of each range and the first (resp. last) statement of each sequence, one obtains the lexical minimum (resp. maximum). An example is found in the next paragraph. Since lexical extrema are so easy to procure at any level of the tree, algorithms that rely on repeatedly computing them are easy to adapt when given a decomposition. A notable example is the building of a finite state machine that indicates, for any leaf statement instance, what the next statement instance is according to the lexicographic order [4]. The non-emptiness of value ranges is here of paramount importance. A full description of the algorithm would take us too far.

Another interesting application of the ability to extract lexical extrema is linear optimization, where the goal is to find inside a polyhedron the integer point(s) that maximizes or minimizes a given function $z = f(x_1, \ldots, x_n)$. The idea is to build a decomposition of an $(n+1)$-dimensional polyhedron, with the z variable placed after/under parameters but before/above regular variables, and start from the logical conjunction of the polyhedron and the function definition. The optimal values are provided by the lexical extreme values on z after decomposition. One can even place congruence requirements on z. For instance, optimizing $15z + 2 = x$ over LS(N) searches for extreme (actually all) values congruent to 2 modulo 15, resulting in:

```
when N = 4
  exec S(1,17,7)
when 5 <= N <= _
  for 5 <= 5z <= N+[N:0,-1,-2,-3,1]
      exec S(z,15z+2,5z+2)
```

For $N < 4$, the polyhedron is too short. Otherwise, the maximum value z^* is such that $5z^* = N - \langle 0,1,2,3,-1\rangle_N$, or:

$$5z^* = N - 3 + \langle 0,4,\ldots,1\rangle_{N-3} = N + 1 - \langle 0,1,\ldots,4\rangle_{N+1}$$

that is, $5z^*$ is *the* multiple of 5 between $N-3$ and $N+1$. The corresponding point is $(15z^*+2, 5z^*+2)$. Note that this example is over-fragmented (again): the case $N = 4$ is covered by the other case.

Finally, the AST is an executable program, modulo syntax. Periodic numbers are trivial to implement, and easy to strength-reduce. The AST-as-a-loop-nest has no branch besides parameter tests and loop control. For reference, we have run (by hand) on LS(N) the pioneering code-generation algorithm of Ancourt and Irigoin [2], which is based on Fourier-Motzkin elimination [1, Chap. 11]. The result is:

```
when 0 <= N <= _
  for 1 <= x <= 3N+7
      for max(⌈$\frac{x+2}{3}$⌉,⌈$\frac{x-N+1}{2}$⌉) <= y <= min(⌊$\frac{x+5}{3}$⌋,⌊$\frac{x+1}{2}$⌋)
          exec S(x,y)
```

All modern code-generators [3, 15] we have tried produce the same result. Code generation is a distinct topic, with different objectives [10], and even defining terms for a meaningful comparison would take us too far.

6 Conclusion

This paper offers a new characterization of linear inequalities over integer variables, with a focus on expressive power and precision. Major operations include tightening, which ensures precise and reversible combinations, and disjoining, which makes inherent disjunctions explicit. Algorithms for decision and decomposition have been described, and some potential applications highlighted. Thanks to the accuracy of periodic-linear expressions, the algorithms are simple repeated applications of elementary operations.

This preliminary work has ignored important questions about complexity (not even considering that almost all problems with integers are NP-hard [12]). First, tightening bounds may produce periodic numbers of size proportional to a coefficient to the power of the number of variables, which is obviously problematic. We expect to alleviate this problem by a combination of laziness (delaying the creation of normalized representations) and arithmetic properties (many examples in this paper have shorter representations: for instance, $\langle 6,1,8,3,4,5\rangle_x \leq x$ is just $\langle 4,1\rangle_x \leq x$). Second, the decomposition algorithm is extremely sensitive to the order in which inequalities are examined, with frequent over-fragmentation, which has a tremendous impact on program size. We expect to avoid these drawbacks with the help of selection heuristics, combined with a dedicated range fusion strategy applicable after damage.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2Nd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[2] Corinne Ancourt and François Irigoin. 1991. Scanning Polyhedra with DO Loops. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '91).* ACM, New York, NY, USA, 39–50.

[3] Cedric Bastoul. 2004. Code Generation in the Polyhedral Model Is Easier Than You Think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT '04).* IEEE Computer Society, Washington, DC, USA, 7–16.

[4] Pierre Boulet and Paul Feautrier. 1998. Scanning Polyhedra Without Do-loops. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT '98).* IEEE Computer Society, Washington, DC, USA, 4–.

[5] Philippe Clauss. 2014. Counting Solutions to Linear and Nonlinear Constraints Through Ehrhart Polynomials: Applications to Analyze and Transform Scientific Programs. In *ACM International Conference on Supercomputing 25th Anniversary Volume.* ACM, New York, NY, USA, 237–244.

[6] Eugène Ehrhart. 1962. Sur les prolyèdres rationnels homothétiques à *n* dimensions. *Comptes-rendus de l'Académie des Sciences* 245 (1962), 616–618. Paris.

[7] Eugène Ehrhart. 1977. *Polynômes arithmétiques et Méthode des Polyèdres en Combinatoire.* Vol. 35. Birkhäuser Verlag, Basel/Stuttgart.

[8] Paul Feautrier. 1988. Parametric integer programming. *RAIRO - Operations Research - Recherche Opérationnelle* 22, 3 (1988), 243–268.

[9] Paul Feautrier and Christian Lengauer. 2011. The Polyhedron Model. In *Encyclopedia of Parallel Computing*, David Padua (Ed.). Springer, 1581–1592.

[10] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. 2015. Polyhedral AST Generation Is More Than Scanning Polyhedra. *ACM Trans. Program. Lang. Syst.* 37, 4, Article 12 (July 2015), 50 pages.

[11] Armin Größlinger, Martin Griebl, and Christian Lengauer. 2006. Quantifier elimination in automatic loop parallelization. *Journal of Symbolic Computation* 41, 11 (2006), 1206 – 1221.

[12] Richard M. Karp. 1972. *Reducibility among Combinatorial Problems.* Springer US, Boston, MA, 85–103.

[13] Benoît Meister. 2004. Periodic Polyhedra. In *Compiler Construction, CC 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain.*, Evelyn Duesterwald (Ed.). Springer, Berlin, Heidelberg, 134–149.

[14] William Pugh. 1991. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing '91).* ACM, New York, NY, USA, 4–13.

[15] Sven Verdoolaege. 2010. ISL: An Integer Set Library for the Polyhedral Model. In *Proceedings of the Third International Congress Conference on Mathematical Software (ICMS'10).* Springer-Verlag, Berlin, Heidelberg, 299–302.

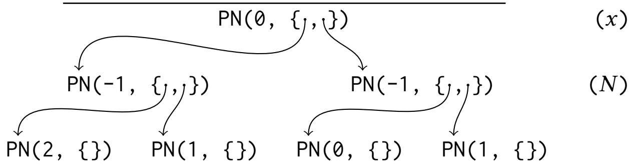
A Implementation Notes

This appendix collects information about some aspects of the implementation of the algorithms described in this paper. It uses a simple C++-based language for code fragments.

A.1 Data Structures

Periodic numbers are defined over an ordered set of variables, and decay into plain integers when that set is empty. Variables names are never stored, and must be provided explicitly when input/input is involved. The data structure for periodic numbers, in Figure 4 (lines 1–8), strictly follows the *simplified normal form* (see Equation (5) in Section 3.1). A periodic number is essentially a tree, with leaves for plain integers and inner nodes carrying a coefficient and a collection of sub-numbers. Here is a graphical depiction of an example number (taken from Figure 2) over variables $[N, x]$:

$$\begin{aligned} & -N + \langle\langle 2,1\rangle_N, \langle 0,1\rangle_N\rangle_x \\ = \quad & 0x + \langle -N + \langle 2,1\rangle_N,\ -N + \langle 0,1\rangle_N\rangle_x \end{aligned}$$



In this paper, periodic numbers have strong invariants: all `coeff` values are identical for a given variable (coefficients are *not* periodic), and so do `phases` size (the constant term is a hyper-rectangle of integers). A dense representation is possible, with a vector of coefficients and a multidimensional array of integers for the constant term. We have favored a generic, sparse representation for ease of implementation, and also to allow the exploration of less constrained settings.

Figure 4 shows the implementation of `add` (lines 9–20), performing addition of two periodic numbers provided they are defined over the same set of variables (`mod` computes arithmetic modulo). Almost all core operations (scaling, gcd extraction and reduction, and so on) follow the same pattern.

A.2 Normalization (Section 3.1)

Most, if not all, of our work relies on the ability to tighten inequalities over integer variables. Tightening takes an arbitrary periodic linear expression and adds a corrective term to "bring it back" to a multiple of a given coefficient: see Equations (6), (7), and (10) in Section 3.2. The corrective term for an PLE X is of the form $\langle 0,1,\dots,\pi-1\rangle^\pi_X$ (or a variant thereof). This section details how such an expression is put in simplified normal form when X itself is.

Assume X is an normalized PLE over variables $[\dots, y, z]$:

$$X = \alpha z + \left\langle Y_0, Y_1, \dots, Y_{\beta-1}\right\rangle^\beta_z$$

where all Y_i, for $0 \le i < \beta$, are PLEs over variables $[\dots, y]$. The rules of Table 1 let us progressively turn the corrective term $\langle 0,\dots\rangle^\pi_X$ into its normal form. Before detailing the

```
1   struct PN {
2       int coeff;
3       vector<PN> phases;
4       PN(int c, vector<PN> p): coeff(c),phases(p) {}
5       int  period () {return phases.size();}
6       bool isplain() {return period() == 0;}
7       PN at(int i) {return phases[mod(i,period())];}
8   };
9   PN add(PN pn1, PN pn2)
10  {
11      assert ( pn1.isplain() == pn2.isplain() );
12      if ( pn1.isplain() ) {
13          return PN(pn1.coeff + pn2.coeff, {});
14      } else {
15          int pi = lcm(pn1.period(), pn2.period());
16          vector<PN> r; // of size pi
17          for ( int i=0 ; i<pi ; i++ )
18              r[i] = add(pn1.at(i), pn2.at(i));
19          return PN(pn1.coeff + pn2.coeff, r);
20  }   }
```

Figure 4. Implementing periodic numbers

derivation, we introduce the following notation:

$$\left\langle\!\!\left\langle \delta \right\rangle\!\!\right\rangle^\pi_X = \langle 0,1,\dots,\pi-1\rangle^\pi_{X+\delta} = {}^i\left\langle \dots, (\delta+i) \bmod \pi, \dots\right\rangle^\pi_X$$

where $0 \le \delta < \pi$. This notation is convenient, because normalization will repeatedly rotate constant periodic numbers.

Here are now the details of the derivation, which is a typical example of symbolic manipulations of periodic numbers. Starting with

$$\langle 0,1,\dots\rangle^\pi_{\alpha z + \langle Y_0,\dots\rangle^\beta_z} \qquad = \left\langle\!\!\left\langle 0 \right\rangle\!\!\right\rangle^\pi_{\alpha z + \langle Y_0,\dots\rangle^\beta_z}$$

$$= {}^i\left\langle \dots, \quad \left\langle\!\!\left\langle i \right\rangle\!\!\right\rangle^\pi_{\langle Y_0,\dots\rangle^\beta_z}, \quad \dots\right\rangle^\pi_{\alpha z} \qquad \text{(after separation)}$$

let $\pi' = \pi/\gcd(\alpha \bmod \pi, \pi)$ and apply division by $\alpha \bmod \pi$

$$= {}^{i'}\left\langle \dots, \quad \left\langle\!\!\left\langle \alpha i' \bmod \pi \right\rangle\!\!\right\rangle^\pi_{\langle Y_0,\dots\rangle^\beta_z}, \quad \dots\right\rangle^{\pi'}_z$$

apply distribution of $\langle Y_0,\dots\rangle^\beta_z$

$$= {}^{i'}\left\langle \dots, \quad {}^j\left\langle \dots, \quad \left\langle\!\!\left\langle \alpha i' \bmod \pi \right\rangle\!\!\right\rangle^\pi_{Y_j}, \quad \dots\right\rangle^\beta_z, \quad \dots\right\rangle^{\pi'}_z$$

let $\pi'' = \operatorname{lcm}(\beta, \pi')$, and apply extension to π'' on the outer periodic number and on all its components; this turns i' into $i'' \bmod \pi'$, which simplifies as

$$= {}^{i''}\left\langle \dots, {}^{j'}\left\langle \dots, \left\langle\!\!\left\langle \alpha i'' \bmod \pi \right\rangle\!\!\right\rangle^\pi_{Y_{(j' \bmod \beta)}}, \dots\right\rangle^{\pi''}_z, \dots\right\rangle^{\pi''}_z$$

and apply diagonalization

$$= {}^k\left\langle \dots, \quad \left\langle\!\!\left\langle \alpha k \bmod \pi \right\rangle\!\!\right\rangle^\pi_{Y_{(k \bmod \beta)}}, \quad \dots\right\rangle^{\pi''}_z \tag{12}$$

This last diagonalization step consists in realizing that two periodic levels have the same argument z and same period π''. These two dimensions coalesce (index k replaces both i'' and j'), and only elements with identical positions at both levels are kept.

Note that the derivation must be slightly generalized to provide a recursive definition of normalization, because Equation (12) contains arbitrarily rotated periodic constant. One can easily verify that:

$$\left\langle\!\left\langle \delta \right\rangle\!\right\rangle^{\pi}_{\alpha z + \langle Y_0,\dots\rangle_z^{\beta}} = {}^{k}\left\langle \dots, \left\langle\!\left\langle (\delta + \alpha k) \bmod \pi \right\rangle\!\right\rangle^{\pi}_{Y_{(k \bmod \beta)}}, \dots \right\rangle^{\pi''}_{z}$$

This last equation can be directly translated into code:

```
PN correction(PN pn, int delta, int pi)
{
    if ( pn.isplain() ) {
        return PN(mod(delta + pn.coeff, pi), {});
    } else {
        int pi1 = pi / gcd(pi, mod(pn.coeff, pi));
        int pi2 = lcm(pn.period(), pi1);
        vector<PN> r; // of size pi2
        for ( int k=0 ; k<pi2 ; k++ ) {
            int delta1 = mod(delta + pn.coeff*k, pi);
            r[k] = correction(pn.at(k), delta1, pi);
        }
        return PN(0, r);
}   }
```

The result of `correction(...)` has the same structure and depth as its first argument, and all coefficients equal to zero.

A.3 Disjoining (Section 3.4)

The Disjoin algorithm turns a *mixed* inequality (where the bound is periodic) over variables $[\dots,x,y,z]$

$$\langle Y_0,\dots\rangle_z^{\pi_z} \le \alpha_z z \qquad \text{with } \pi_z > 1 \tag{13}$$

into a logical expression where all bounds are *no more* periodic in z ($A = B$ is a shorthand for $A \le B \wedge B \le A$):

$$(\alpha_z z = O_k) \vee \dots \vee (\alpha_z z = O_1) \vee (M \le \alpha_z z)$$

All of M (the *major* bound) and O_j ($1 \le j \le k$, the *outliers*) are PLEs over $[\dots,y]$, not involving z anymore. We will henceforth use notation $\{M, O_1, \dots, O_k\}$ to collectively represent the major bound and all outliers.

The principles of disjoining multidimensional bounds is illustrated on the following picture:

$$\begin{tikzcd}
\left[\begin{bmatrix}X_{00}\\ X_{01}\\ \vdots\end{bmatrix}_y^{\pi_y}, \begin{bmatrix}X_{10}\\ X_{11}\\ \vdots\end{bmatrix}_y^{\pi_y}, \dots\right]_z^{\pi_z} \arrow[r, "(a)"] & \begin{bmatrix}{[X_{00},X_{10},\dots]_z^{\pi_z}}\\ {[X_{01},X_{11},\dots]_z^{\pi_z}}\\ \vdots\end{bmatrix}_y^{\pi_y} \arrow[d, dashed, bend left, "(b)"] \\
\left\{\begin{bmatrix}M_0\\ M_1\\ \vdots\end{bmatrix}_y^{\pi_y}, \begin{bmatrix}O_{01}\\ O_{11}\\ \vdots\end{bmatrix}_y^{\pi_y}, \dots\right\} & \begin{bmatrix}\{M_0,O_{01},\dots\}\\ \{M_1,O_{11},\dots\}\\ \vdots\end{bmatrix}_y^{\pi_y} \arrow[l, "(c)"']
\end{tikzcd}$$

Starting from a bound with $\pi_z > 1$, step (a) does an interchange of the outer two variables, pulling y out to represent the bound as phases of y modulo π_y. Step (b) is a recursive processing of these phases, acting over the remaining variables $[\dots,x,z]$, and returning a complete set of bounds for each phase of y. Step (c) turns a periodic series of lists of bounds into a list of periodic bounds.

Overall, the processing is surrounded by two transpositions, and recurses until reaching a situation where z is the only remaining variable; at this point, Disjoin_1 (see Section 3.4) applies. This happens for every possible phase of variables $[\dots,x,y]$. A simplified version of the code is:

```
vector<PN> disjoin(vector<PN> ps, int alpha)
{
    int piz = ps.size();
    if ( ps[0].isplain() ) {
        int vm = ps[0].coeff;
        for ( int i=1 ; i<piz ; i++ )
            vm = max(vm, ps[i].coeff);
        int M = vm - alpha*(piz-1);
        vector<PN> bounds; // initially empty
        bounds.append(PN(M, {}));
        for ( int i=0 ; i<piz ; i++ )
            if ( ps[i].coeff < M )
                bounds.append(PN(ps[i].coeff, {}));
        return bounds;
    } else {
        int piy = ps[0].period();
        vector<vector<PN>> rec; // of size piy
        for ( int j=0 ; j<piy ; j++ ) {        // (a)
            vector<PN> cj; // of size piz
            for ( int i=0 ; i<piz ; i++ )
                cj[i] = ps[i].phases[j];
            rec[j] = disjoin(cj, alpha);       // (b)
        }
        // ... standardize sizes in rec ...
        int bsize = rec[0].size();
        vector<PN> bounds; // of size bsize
        for ( int k=0 ; k<bsize ; k++ ) {      // (c)
            vector<PN> ph; // of size piy
            for ( int j=0 ; j<piy ; j++ )
                ph[j] = rec[j][k];
            bounds[k] = PN(ps[0].coeff, ph);
        }
        return bounds;
}   }
```

Given an inequality such as Equation (13), the initial call to `disjoin` is with arguments $[Y_0,\dots]$ and α_z. This code relies on periodic number invariants mentioned earlier: coefficients must not be periodic, and bounds must be rectangular (all π_y values equal in the schema above). To simplify exposition, it assumes that all lists of bounds are of the same size (variable `bsize`): in practice, the lengths may need to be standardized by adding dummy bounds. Also ignored is the fact that outliers can be grouped into intervals. In general, the treatment of outliers supports several variations.