

Easy Counting and Ranking for Simple Loops

Alain Ketterlin

Inria/Camus, CNRS/Icube, Université de Strasbourg (France)
alain@unistra.fr

Abstract

This paper describes a way to count instructions executed by a loop nest, and assign a sequence number to each one of them. It is restricted to a specific class of affine loop nests, and uses an uncompromising representation of integer polynomials, which is described in detail. The general focus is on mathematical and algorithmic simplicity, searching for easy to implement methods under acceptable restrictions.

1 Motivation

The polyhedral model establishes a correspondence between sets of integer points delimited by hyperplanes and various characteristics of affine loop nests, such as iteration domains, all sorts of dependencies, or scheduling functions [5]. It excels at representing and transforming multi-dimensional integer sets, and has been used extensively in a variety of compilation tasks [8]. Much less studied are the quantitative aspects of polyhedra and/or the corresponding loop nests.

Counting [1, 15] consists in determining a formula representing the number of integer points inside a bounded polyhedron, as a function of the parameters appearing in the polyhedron’s description. The result can be used for complexity analysis, to predict various types of performance metrics, or as heuristics to guide optimization.

Ranking consists in assigning a unique, sequentially incremented number to each integer point inside a bounded polyhedron. As such, it only makes sense with respect to a schedule, when the traversal order has been fixed. The ranking function can be seen as a “flat” version of the schedule. The inverse action, called *unranking*, transforms a sequential index into a multi-dimensional iteration vector. It essentially provides random access to instruction instances, or slices thereof, for example for sampling or parallel execution.

The counting problem has been solved in a general polyhedral setting, in two different ways. Clauss [1], building on work by Ehrhart, has formulated a solution producing Ehrhart pseudo-polynomials, and Verdoolaege et al. [15], building on work by Barvinok, have given a solution producing step-polynomials. In both cases, the mathematical concepts involved are far from trivial, the result has a (necessarily) unusual form, and the algorithms are complex. Defined on top of counting, ranking also needs an additional scheduling step, which often makes it application-specific. Therefore, these remarkable scientific achievements have found little application beyond counting.

Our approach in this paper is to offer alternative counting, ranking and unranking algorithms that are conceptually and technically simple and allow lightweight and fast implementations. We achieve this by sacrificing generality. First and foremost, the algorithms operate on loops, not unrestricted polyhedra. Second, the general approach is similar in spirit to that pioneered by Pugh [12] (which was also targeting polyhedra, but was apparently never fully implemented). It consists in considering an isomorphism between a loop nest and an algebraic expression of the count of atomic instructions the loop nest executes. Counting a loop amounts to translating it into a sum, in a syntactic way, and then applying algebraic manipulations to put the result in a form that is suitable for further processing.

The running example in this paper is the Cholesky kernel from the Polybench suite [11] version 3. It is shown below, along with its translation into a counting expression (note that, by convention, upper bounds are always excluded from the iteration range):

$$\begin{array}{ll} \text{for } i = 0 \text{ to } n & \sum_{i=0}^{n-1} (\\ \text{S1} & 1 \\ \text{for } j = 0 \text{ to } i & + \sum_{j=0}^{i-1} (\\ \text{S2} & 1) \\ \text{S3} & +1 \\ \text{for } j = i+1 \text{ to } n & + \sum_{j=i+1}^{n-1} (\\ \text{S4} & 1 \\ \text{for } k = 0 \text{ to } i & + \sum_{k=0}^{i-1} (\\ \text{S5} & 1) \\ \text{S6} & +1)) \end{array}$$

After translation, the expression on the right-hand side can be turned into whatever form is most convenient, typically a polynomial. Once the counts (at all levels) are computed, they serve as a basis to compute ranks, which in turn are used for unranking.

This paper goes through the details of this process. Section 2 lists the conditions a loop nest must respect for its transformation into a sum to have a meaning, and briefly explains how to enforce them. Section 3 introduces an exotic representation of integer polynomials (*not* a representation of exotic polynomials, for once); this representation makes it easy to build sums like the above and transform them into a standard form of multivariate polynomials. Section 4 describes counting, ranking, and unranking algorithms, and gives detailed results for the previous example. The paper ends by discussing the importance of the trade-offs conceded, speculating on some applications, and pointing to a straightforward implementation of everything that is exposed.

2 Simple Loops

We consider programs made only of loops and basic instructions, collectively called *statements*. Every loop has a lower bound and an upper bound, and a body that is an arbitrary (but non-empty) sequence of statements. A basic instruction will be represented by an identifier, and its details are immaterial. A full program has a list of constant symbolic parameters, a context, which is a set of conditions on the values of the parameters, and a sequence of statements.

Variables are either parameters or loop counters. The scope of a parameter is the entire program, while the scope of a loop counter is the body of the loop that defines it. The context is made of affine inequalities involving only the parameters. Loop bounds are affine expressions in the variables in scope at the start of the loop. No integer function like min or max, or any form of integer part, is allowed.

For the counting strategy outlined earlier to be valid, loops have to respect certain criteria. Because loops are turned into sums, there must be a perfect match between the operational meaning of a loop and the mathematical semantics of the resulting sum. Since we have adopted the convention that loops iterate from their lower bound included to their upper bound excluded, counting works by applying the rule:

$$\text{for } i=l \text{ to } u \dots \Rightarrow \sum_{i=l}^{u-1} \dots$$

This only makes sense if the input loop is *simple*, which here means it respects the following *validity* constraints:

1. *unit step*: the loop counter is incremented by 1 from one iteration to the next;
2. *bounds coherence*: every execution of the loop must happen in a situation where $l \leq u$.

Bounds coherence essentially means that every instance of a loop must “do something”. However, it also tolerates a loop doing *exactly nothing* (when $l = u$), thanks to a mathematical peculiarity that makes a sum of the form $\sum_a^{a-1} \dots$ resolve to a meaningful result (namely, zero)—see Section 3.2. Such unproductive loop instances are not rare in practice, because they allow for simpler logic and shorter code; every single inner loop of the Cholesky kernel in Section 1 has one.

The first constraint can be enforced syntactically by forbidding a non-unit iteration step. The second constraint however needs explicit verification, because it applies to every instance of every loop. For instance, given:

```
for i=0 to 10
  for j=0 to 5 - i
    . . .
```

the loop on j is invalid, because *some* of its instances are absurd (those for $6 \leq i < 10$).

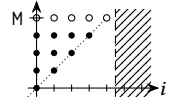
Bounds coherence must be verified for each loop in the program. For a given loop with bounds l and u , both of which being affine combinations of variables in scope, the verification procedure proceeds by collecting the system of inequalities that defines the problematic loop instances:

1. constraints on parameters (the context);
2. inequalities on the bounds of enclosing loops;
3. the inequality $u < l$, signaling an absurd instance.

This system of inequalities must then be proven false. In the previous example, the system is $\{0 \leq i, i < 10, 5 - i < 0\}$: since this system has solutions, the loop is invalid. There are many ways to perform this test: an easy one is repeated Fourier-Motzkin elimination of variables and parameters until *false* is inferred (or not), which may be inexact but conservative.

Here is one last example illustrating the role of the context, or absence thereof, about two parameters N and M :

```
for i=0 to N
  for j=i to M
    . . .
```



First, the outer loop is invalid unless $N \geq 0$, and this condition (or a stronger condition on N) must appear in the context. Second, testing the validity of the inner loop amounts to testing the vacuity of $\{0 \leq i, i < N, M < i\}$, which after elimination of i leads to $M < N - 1$, from which the loop is declared invalid in the absence of more context. Adding $M \geq N - 1$ in the context would make the loop valid. Note that when $M < N - 1$ actually holds, the outer loop needs to be rewritten as “for $i=0$ to $M \dots$ ” to be considered valid; there is no valid program covering both cases under our definition of validity.

3 Integer Polynomials

It is known from previous work that the number of integer points inside a polyhedron (or a union of polyhedra) can be represented by a polynomial, or rather by some extended representation, introducing either periodic numbers [1] or making use of integer-parts [15]. This section shows that counting simple loops does not require these extensions. It also introduces an alternative representation of integer polynomials (in general, not specific to counting), for the sake of precision and simpler algebra.


3.1 Representation

A univariate integer polynomial is an integer-valued polynomial in an integer variable. Integer polynomials suffer from a representation mismatch when using “regular” powers $\{x^k \mid k \geq 0\}$. Combining these monomials with integer coefficients leads to an incomplete representation; for instance, $x(x - 1)/2$ cannot be represented. Unfortunately, using rational coefficients leads to an incorrect representation; for instance, $x(x - 2)/2$ is not integer-valued.

We will use an alternative monomial basis $\{x^{\lfloor k \rfloor} \mid k \geq 0\}$ instead, where $x^{\lfloor k \rfloor}$ is defined as:

$$x^{\lfloor k \rfloor} \triangleq \binom{x}{k} = \frac{x \cdot (x - 1) \cdots (x - k + 1)}{k!}$$

We use this notation to emphasize the fact that binomial coefficients, extended to negative x , act as powers: $x^{\lfloor k \rfloor}$ is defined for any x , and any non-negative k . Additionally, $x^{\lfloor k \rfloor}$ has a myriad of properties associated with binomial coefficients, the most important of which is Pascal's identity:

$$(x + 1)^{\lfloor k+1 \rfloor} = x^{\lfloor k \rfloor} + x^{\lfloor k+1 \rfloor}$$


which also holds for negative x [7, Eq. 5.14]. All binomial powers reside in Pascal's extended triangle; the figure shows a graphical equivalent of the identity. The relation between regular and binomial powers is known:

$$x^{\lfloor n \rfloor} = \frac{1}{n!} \sum_{k=0}^n (-1)^{n-k} \begin{bmatrix} n \\ k \end{bmatrix} x^k \quad x^n = \sum_{k=0}^n k! \begin{Bmatrix} n \\ k \end{Bmatrix} x^{\lfloor k \rfloor}$$

where $\begin{bmatrix} n \\ k \end{bmatrix}$ and $\begin{Bmatrix} n \\ k \end{Bmatrix}$ are the unsigned Stirling numbers of the first and second kind. Regular powers will not be used further here, nor will rational numbers. In the rest of this paper, the term *polynomial* refers to combinations of binomial powers with integer coefficients. (In mathematical parlance, regular integer polynomials are represented with their Newton series expansion [7, Sec. 5.3], which is then used for all purposes.)

This representation is correct and complete for integer polynomials. Correctness follows from the fact that only integers are involved, but completeness requires a proof; we sketch it here because it also forms the core of algebraic manipulations described later in Section 3.4. Since integer polynomials produce integer values, we need to prove that any sequence of integers v_0, \dots, v_n can be produced by a polynomial of appropriate degree under our definition. Noting $p(x) = a_0 + a_1 x^{\lfloor 1 \rfloor} + \dots + a_n x^{\lfloor n \rfloor}$, where coefficients a_i are unknown, the sequence of values is produced by $p(x)$ if:

$$\begin{aligned} v_0 &= p(0) = a_0 \\ v_1 &= p(1) = a_0 + a_1 \cdot 1^{\lfloor 1 \rfloor} \\ v_2 &= p(2) = a_0 + a_1 \cdot 2^{\lfloor 1 \rfloor} + a_2 \cdot 2^{\lfloor 2 \rfloor} \end{aligned}$$

and so on until v_n . Since $i^{\lfloor k \rfloor} = 0$ when $0 \leq i < k$ this system is triangular, and since $i^{\lfloor i \rfloor} = 1$ it always has a unique integer solution. This proves completeness, in a constructive way: the solution can be computed incrementally as:

$$a_0 = v_0, \quad a_i = v_i - \sum_{j=0}^{i-1} a_j \cdot i^{\lfloor j \rfloor} \quad (i > 0)$$

or directly from the values as:

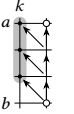
$$a_i = \sum_{j=0}^i (-1)^{i-\lfloor j \rfloor} \cdot i^{\lfloor j \rfloor} \cdot v_j$$

where $(-1)^{i-\lfloor j \rfloor}$ is used for consistency: it is equal to $(-1)^{i-j}$. The proof of the last set of equalities (omitted here) is a simple induction using the previous, or by noting that the initial system can be written $\vec{v} = \mathcal{P}_{n+1} \vec{a}$, where \mathcal{P}_{n+1} is a matrix whose lower triangular part is equal to the first $n + 1$ lines of Pascal's triangle and other elements are zero. The solution is obtained by multiplying both sides by \mathcal{P}_{n+1}^{-1} .

3.2 Sums of Powers and Polynomials

Counting loops requires addition and summation of polynomials over an interval. Adding two polynomials is done as usual by adding coefficients of identical monomials.

Summing a polynomial over an interval relies on the following definition of sums of (binomial) powers:

$$\sum_{x=a}^{b-1} x^{\lfloor k \rfloor} = b^{\lfloor k+1 \rfloor} - a^{\lfloor k+1 \rfloor} \quad (a < b)$$


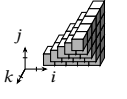
The figure illustrates the proof strategy: starting with $b^{\lfloor k+1 \rfloor}$, every application of Pascal's identity sets aside one term of the sum, and leaves another term that is closer to $a^{\lfloor k+1 \rfloor}$.

We can relax the condition $a < b$ by adopting the convention that $\sum_{x=a}^{a-1} x^{\lfloor k \rfloor} = 0$, and use the identity whenever $a \leq b$, which explains our definition of simple loops in Section 2. Incidentally, the special case $a = 0$ gives an insight into the meaning of binomial powers: intuitively, triangles are to $x^{\lfloor k \rfloor}$ what squares are to x^k , because, under our relaxed setting

$$\sum_{i_1=0}^{n-1} \sum_{i_2=0}^{i_1-1} \dots \sum_{i_d=0}^{i_{d-1}-1} 1 = n^{\lfloor d \rfloor}$$

and here is their interpretation in terms of loop counting (where with/when introduce a parameter and the context):

| | | | |
|--------------------------|---------|-----------|-------------------------|
| with n when $n \geq 0$ | (count) | $(n = 7)$ | |
| for $i=0$ to n | | | $n^{\lfloor 3 \rfloor}$ |
| for $j=0$ to i | | | $i^{\lfloor 2 \rfloor}$ |
| for $k=0$ to j | | | $j^{\lfloor 1 \rfloor}$ |
| S | | | 1 |



This extremely simple summation rule for monomials extends to polynomials as follows. Given a polynomial

$$p(x) = \sum_{i=0}^n a_i \cdot x^{\lfloor i \rfloor}$$

its anti-difference, also called indefinite sum [7, Eq. 2.46], with constant 0 is

$$\Delta^{-1} p(x) = \sum_{i=0}^n a_i \cdot x^{\lfloor i+1 \rfloor}$$

and summing $p(x)$ when x ranges over $[a, b)$ gives

$$\sum_{x=a}^{b-1} p(x) = \Delta^{-1} p(x) \Big|_a^b = \Delta^{-1} p(b) - \Delta^{-1} p(a)$$

That's all there is. In practice, $\Delta^{-1} p(x)$ is obtained by simply incrementing all exponents of $p(x)$ or, equivalently, shifting its coefficients. For instance, given the following loop whose body has a known per-iteration count:

$$\begin{aligned} &\text{for } i=0 \text{ to } N \\ &\quad \dots \text{ (executes } 3i^{\lfloor 1 \rfloor} + 7i^{\lfloor 2 \rfloor} \text{ instructions)} \end{aligned}$$

one can immediately assert that the loop executes a grand total of $3N^{\lfloor 2 \rfloor} + 7N^{\lfloor 3 \rfloor}$ instructions; if the lower bound were 5 instead of 0, it would execute $3 \cdot 5^{\lfloor 2 \rfloor} + 7 \cdot 5^{\lfloor 3 \rfloor} = 100$ fewer instructions (assuming the loop is valid in both cases).

3.3 Multivariate Polynomials

The univariate polynomial framework just described must now be extended to polynomials in multiple variables, because counting and ranking polynomials may use all variables in scope at a given program point. Throughout this section, the following abstract pattern will be used:

| | |
|--|----------------|
| with n | (count) |
| L_i : for $i=l_i(n)$ to $u_i(n)$ | $\#L_i(n)$ |
| L_j : for $j=l_j(n, i)$ to $u_j(n, i)$ | $\#L_j(n, i)$ |
| B : . . . | $\#B(n, i, j)$ |

Now, if $\#B$ is the number of instruction executions of an instance of B , then this quantity is a polynomial in n , i , and j , since all of these variables are in scope and may appear in loop bounds controlling the details of the execution of B .

A concrete example, namely the second loop on j in the Cholesky kernel from Section 1, will also serve as illustration, where $\#B$ is known, via unspecified means at this point:

| | |
|----------------------------|------------------------|
| . . . | (count) |
| L_j : for $j=i+1$ to n | $\#L_j(n, i)$ |
| B : . . . | $\#B(n, i, j) = 2 + i$ |

This section details how such multivariate polynomials are represented and manipulated.

Let $\mathbb{Z}[x_1, \dots, x_d]$ be the set of multivariate polynomials in d variables. An element of $\mathbb{Z}[x_1, \dots, x_d]$ is

$$p(x_1, \dots, x_d) = \begin{cases} a_0 & \text{if } d = 0 \ (a_0 \in \mathbb{Z}) \\ \sum_{i=0}^r a_i x_d^i & \text{if } d > 0 \ (a_i \in \mathbb{Z}[x_1, \dots, x_{d-1}]) \end{cases}$$

In the second case, p has degree r w.r.t. x_d . We will use this inductive definition literally, and consider polynomials factored along one variable at a time. In our previous abstract pattern, we would have:

$$\#B(n, i, j) = a_0(n, i) \cdot j^{0j} + a_1(n, i) \cdot j^{1j} + \dots + a_r(n, i) \cdot j^{rj}$$

i.e., a polynomial in j whose coefficients are polynomials in n and i , the first of which can be written as:

$$a_0(n, i) = b_{0,0}(n) \cdot i^{0i} + \dots + b_{0,r_0}(n) \cdot i^{r_0i}$$

and others are similar, possibly with varying degrees $r_0, r_1 \dots$. In our concrete example, the full expressions of the various polynomials, omitting terms with zero coefficient, are:

$$\begin{aligned} l_j(n, i) &= ((1) \cdot n^{0i}) \cdot i^{0i} + ((1) \cdot n^{0i}) \cdot i^{1i} \\ u_j(n, i) &= ((1) \cdot n^{1i}) \cdot i^{0i} \\ \#B(n, i, j) &= (((2) \cdot n^{0i}) \cdot i^{0i} + ((1) \cdot n^{0i}) \cdot i^{1i}) \cdot j^{0i} \end{aligned}$$

We will dispense with these details below. But overall, a multivariate polynomial can be seen as a tree with one level per variable, with coefficients on leaves, and with every path from the root representing one particular combination of exponents on the various variables. We use this as a *standard form* for polynomials, with variables consistently used in the reverse order of their introduction.

In the abstract pattern above, with $\#B(n, i, j)$ in standard form, the computation of $\#L_j(n, i)$ proceeds as follows:

$$\begin{aligned} \#L_j(n, i) &= \sum_{j=l_j(n, i)}^{u_j(n, i)-1} \#B(n, i, j) \\ &= \sum_{j=l_j(n, i)}^{u_j(n, i)-1} \sum_{k=0}^r a_k(n, i) \cdot j^{kj} \\ &= \sum_{k=0}^r a_k(n, i) \cdot j^{k+1j} \Big|_{l_j(n, i)}^{u_j(n, i)} \\ &= \sum_{k=0}^r a_k(n, i) \cdot \left(u_j(n, i)^{k+1j} - l_j(n, i)^{k+1j} \right) \end{aligned}$$

The result is, as expected, an expression in n and i , involving loop bounds $l_j(n, i)$ and $u_j(n, i)$, both affine functions (polynomials of degree at most 1), and the polynomials $a_k(n, i)$ coming from $\#B(n, i, j)$. In the concrete example, this computation gives:

$$\begin{aligned} \#L_j(n, i) &= \sum_{j=i+1}^{n-1} (2+i) = (2+i) \cdot j^{1j} \Big|_{i+1}^n \\ &= (2+i)(n-i-1) \end{aligned}$$

In general, the result involves exponentiation and multiplication of various polynomials. A bit more effort is needed to put such expressions in standard form (factored along powers of i), so that the resulting polynomial can itself be subjected to summation over i to compute $\#L_i(n)$.

3.4 Algebraic Operations

Binomial powers follow their own algebraic rules, which may be surprising at times: for instance $x^{1j} \cdot x^{1j}$ is equal to $2x^{2j} + x^{1j}$, and $(x^{2j})^{2j}$ is equal to $3x^{4j} + 3x^{3j}$. There is however a powerful strategy to put an arbitrary expression into standard form: interpolation. Given an arbitrary expression $e(x)$ in a variable x , made of polynomials and additions, multiplications and exponentiations thereof, it can be put in standard form with respect to x by evaluating it on a sufficient number of small integers and applying the interpolation formula from Section 3.1 on the results:

$$e(x) = \sum_{i=0}^{|e(x)|} x^{ij} \cdot \left(\sum_{j=0}^i (-1)^{i-j} \cdot i^{ij} \cdot e(j) \right)$$

The bound $|e(x)|$ is the degree of $e(x)$ with respect to x . The rules to estimate it are identical to those governing regular polynomials: $|e_1 \cdot e_2| = |e_1| + |e_2|$, etc.

The concrete example of the previous section found that $\#L_j(n, i) = (2+i)(n-i-1)$. We can now factor this expression along powers of i , by noting it has degree 2, hence evaluating it for $i = 0, 1, 2$ and combining the results:

$$\begin{array}{ccc}
\#L_j(n, 0) & \#L_j(n, 1) & \#L_j(n, 2) \\
= 2(n-1) & = 3(n-2) & = 4(n-3) \\
\downarrow +1 & \swarrow -1 \quad \downarrow +1 & \swarrow -2 \quad \downarrow +1 \\
(2n-2) \cdot i^{0j} & + (n-4) \cdot i^{1j} & -2 \cdot i^{2j}
\end{array}$$

We have waved a magic wand here to obtain the final form of the coefficients, which are polynomials in n . As we will see shortly, they are also obtained by interpolation.

Applying the interpolation strategy to multivariate expressions amounts to interpolating along each variable in turn. Noting σ the standardization operation, we have:

$$\sigma(e(x_1, \dots, x_d)) = \sum_{i=0}^{|e|_d} x_d^i \cdot \left(\sum_{j=0}^i (-1)^{i-j} \cdot i^{j|} \cdot \boxed{\sigma(e(x_1, \dots, x_d \mapsto j))} \right)$$

The operand is an expression in d variables, only the last one of which is subject to interpolation. Here $|e|_d$ is the degree of e with respect to the interpolated variable x_d only. This is an inductive definition, the boxed part being the inductive step, where the notation $e(\dots, x_d \mapsto j)$ represents an expression in $d-1$ variables obtained by substituting the number j for x_d . This bottoms out with an expression in no variable, i.e., a number. The final result is in standard form.

The example above has omitted to show three such recursive calls, to turn $2(n-1)$, $3(n-2)$ and $4(n-3)$ into standard forms before linear combinations. Here is what happened with, e.g., $2 \cdot (n-1)$, an expression of degree 1:

$$\begin{array}{ccc}
2 \cdot (0-1) & & 2 \cdot (1-1) \\
\downarrow +1 & \swarrow -1 & \downarrow +1 \\
-2 \cdot n^{0j} & & +2 \cdot n^{1j}
\end{array}$$

This less-than-impressive example should not hide the fact that specializing σ for basic algebraic operations keeps multivariate polynomials in standard form at all times.

Everything is now in place to implement the counting strategy outlined in Section 1. Before moving on, note that the counting machinery just presented has no trouble handling polynomial loop bounds. For instance, the reader may be interested to learn that the trivially valid program

```

with n m when n ≥ 0 and m ≥ 0
  for i=0 to n2j
    for j=0 to n · i2j + m · i
      S

```

executes instruction S exactly

$$3(m+1)n^{3j} + (68+3m)n^{4j} + 230n^{5j} + 270n^{6j} + 105n^{7j}$$

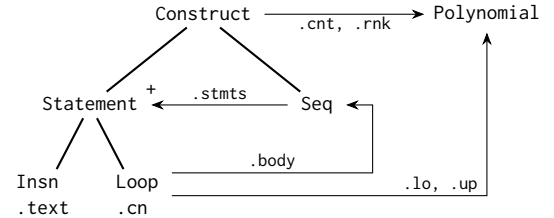
times. Nevertheless, affine bounds remain a requirement for simple loops, because we know of no general method to verify bounds coherence in the polynomial case, and have deferred to future work, should the need arise, the choice of an appropriate validation technique.

4 Counting and Ranking (and back)

Given the definition of simple loops in Section 2, and the polynomial representation just described, this section turns to algorithms that compute counts and ranks according to the strategy described in Section 1. The stated goal here is simplicity, and this section actually looks like a collection of textbook exercises in tree traversal. Python syntax is used for algorithms, but some details have been abstracted away, or plainly omitted—see Section 5.4.

4.1 Building Blocks

Programs are represented as abstract syntax trees; the various types of constructs and their relations are shown on the following graph, with arrows carrying attribute names. The body of a loop is of type Seq, representing a sequence of either atomic instructions or sub-loops; this strict Loop/Seq alternation makes it easy to locate any given program point. Attributes `text` and `cn` (the name of the counter) play no significant role and are used only in displays.



Loop bounds `lo` and `up` are polynomials, even though they must be affine. Every program construct has two attributes `cnt` and `rnk` that have no initial value, and are set by the functions below. Details of the `Polynomial` type are omitted; such objects are manipulated via the following functions:

- value (p, \vec{v}): computes $p(\vec{v})$ (a number);
- add/sub (p, q): polynomial addition/subtraction;
- adiff (c, p): builds $\Delta^{-1}p$ with constant c ;
- sumto (p, b): builds $\Delta^{-1}p(b)$ (with constant 0).

Note that it takes two calls to `sumto` to implement the summation formula described in Section 3.2.

4.2 Counts and Ranks

Counting instruction instances proceeds bottom-up, adding up sub-constructs' counts to obtain a construct's total count.

```

1 def count (c):
2   if c.isseq ():
3     c.cnt = 0
4     for s in c.stmts:
5       count (s)
6     c.cnt = add (c.cnt, s.cnt)
7   elif c.isloop ():
8     count (c.body)
9     c.cnt = sub (sumto (c.body.cnt, c.up),
10                sumto (c.body.cnt, c.lo))
11  else: # insn
12    c.cnt = 1

```

Methods `isseq()` and `isloop()` discriminate between subtypes of Construct and Statement. The misleadingly simple constants 0 and 1 in this fragment are in fact multivariate polynomials in whatever variables are in scope. Instructions could be assigned any weight other than 1 as long as it is expressed as a positive polynomial in the variables in scope.

There are several possible strategies to assign a rank to each and every program point. The one described below is straightforward, and uses pre-computed counts. It proceeds top-down and, when acting on a given construct, computes ranks for all constructs immediately underneath it. As such, there is nothing left to do when reaching an instruction, and the root construct must be explicitly primed with a null rank:

```

1 def rank (c):
2     c.rnk = 0
3     rank_aux (c)
4 def rank_aux (c):
5     if c.isseq ():
6         r = c.rnk
7         for s in c.stmts:
8             s.rnk = r
9             rank_aux (s)
10            r = add (r, s.cnt)
11     elif c.isloop ():
12         a0 = sub (c.rnk, sumto (c.body.cnt, c.lo))
13         c.body.rnk = adiff (a0, c.body.cnt)
14         rank_aux (c.body)

```

When handed a sequence, this algorithm simply walks the statements, accumulating instruction counts to assign ranks. When handed a loop, here is the situation:

```

c:           for i=l to u
c.body:     . . . (.cnt=b0+b1i1+...+bnin)

```

The rank of the start of iteration i is

$$c.rnk + \Delta^{-1}c.body.cnt(i) - \Delta^{-1}c.body.cnt(l)$$

that is, the rank of the start of the loop plus the accumulated count of iterations $l, \dots, i-1$. Only the second term depends on i ; expanding it and rearranging leads to:

$$[c.rnk - \Delta^{-1}c.body.cnt(l)] + b_0i^1 + b_1i^2 + \dots + b_ni^{n+1}$$

i.e., a term (between brackets) that is constant with respect to i , plus the anti-difference $\Delta^{-1}c.body.cnt$ with constant 0. The `adiff` function simply assembles both parts.

Figure 1 shows counts and ranks for the Cholesky kernel.

4.3 Rank Inversion

The collection or ranking polynomials assigns a unique linear number to every instruction execution. This section examines the inverse problem: given a numeric rank, determine the corresponding instruction and the counter values of its enclosing loops. Clauss et al. [2] have presented a symbolic solution to this very problem, which can therefore potentially be used at compile-time. The solution presented here is different in that it relies directly on numerical resolution, and eventually produces code computing the solution.

The first algorithm takes as input an AST, the values of the parameters, and a numeric rank, and finds its way down the AST using ranking polynomials along the way. Every time it reaches a construct, it has already determined the values of all the variables in scope, and can therefore decide which sub-construct contains the target instruction instance. When reaching a sequence of statements, this means comparing the input rank to the ranks of the various elements of the sequence. When reaching a loop, this means determining which iteration of that loop executes the target instruction. During the search, the algorithm collects (separately) the positions of sequence elements and the iteration numbers of loops it traverses. In the code below, `c` is a construct, `vs` the list of known variable values, `ps` the list of known sequence positions, and `r` the target rank (when called on the root construct, `vs` contains parameter values and `ps` is empty).

```

1 def unrank (c, vs, ps, r):
2     if c.isseq ():
3         pos = 0
4         while (pos+1 < len (c.stmts) and
5             r >= value (c.stmts[pos+1].rnk, vs)):
6             pos += 1
7         return unrank (c.stmts[pos], vs, ps+[pos], r)
8     elif c.isloop ():
9         urnk = uni (c.body.rnk, vs)
10        lo = value (c.lo, vs)
11        up = value (c.up, vs)
12        index = unisolve (urnk, lo, up, r)
13        return unrank (c.body, vs+[index], ps, r)
14    else:
15        return (vs, ps)

```

When reaching the target instruction, `vs` and `ps` provide a unique instruction instance identification; a single iteration vector can be recovered by interleaving these two vectors. Note that the given rank r *must* correspond to an actual instruction instance.

The block dealing with loops uses two new functions:

`uni` (p, \vec{v}): computes the numerical values of the coefficients of p with arguments \vec{v} , returning a univariate polynomial whose sole unknown is the loop counter;
`unisolve` (p, l, u, r): computes the largest value i in $[l, u)$ such that $p(i) \leq r$, i.e., the iteration executing the instruction with rank r .

Besides possible special cases for low degree, here is a generic implementation for `unisolve`, using arithmetic bisection:

```

1 def unisolve (pol, lo, up, r):
2     mi = (lo + up) // 2
3     while lo < mi:
4         if value (pol, [mi]) > r:
5             up = mi
6         else:
7             lo = mi
8     mi = (lo + up) // 2
9     return lo

```

Note that the direction of the test against r guarantees the correct result, even in the presence of unproductive iterations. Using other root-finding strategies is certainly possible, keeping in mind that the polynomial may have any kind of behavior outside the given interval.

The unrank algorithm is suitable for situations where the AST is available, which may not be the case in a compiled program. Since unrank follows a single path down a tree of finite size, it is a simple matter to generate a complete symbolic tree traversal instead, where every test is output instead of being performed, and every expression is printed instead of being evaluated. We give here only a sketch of the corresponding function to illustrate the basic principle. Parameter `ns` is the list of names in scope, `c` is a construct, and `ps` is a locator for `c` (as a list of positions in sequences):

```

1 def unranker (ns, c, ps):
2   if c.isseq ():
3     for i in range (len (c.stmts)-1):
4       p = expr (c.stmts[i+1].rnk, ns)
5       print ("if RANK < %s:" % (p))
6       unranker (ns, c.stmts[i], ps+[i])
7     . . .
8   elif c.isloop ():
9     urnk = uexpr (c.body.rnk, ns)
10    lo = expr (c.lo, ns)
11    up = expr (c.up, ns)
12    print ("%s = unisolve (%s, %s, %s, RANK)"
13          % (c.cn, urnk, lo, up))
14    unranker (ns+[c.cn], c.body, ps)
15  else:
16    print ("return (%s, %s)" % (ns, ps))

```

Functions `expr` and `uexpr` produce a compilable text version of the given polynomial. The real code of `unranker` is uninterestingly convoluted because it has to produce correct `if/elif/else` cascades of tests, and has other pretty printing duties. The resulting output code introduces new variables for computed loop indices; by construction, name definitions are guaranteed to dominate their uses. The logic must also be wrapped in a function accepting parameter values and the target rank (called `RANK` above), and a version of `unisolve` must be shipped along.

Figure 2 shows the output code for the Cholesky kernel. Unranking happens at run-time, and `unisolve` is clearly the most expensive operation. Because ranking polynomials are often linear in the nearest loop counter, some of its calls can be avoided and the solution be computed directly. Figure 3 demonstrates this trivial optimization on a simple example.

5 Discussion

This section closes the paper by looking back at the conditions making loops acceptable to the algorithms just described, and contrasting those with earlier work. It also tries to briefly review some existing applications, and to draw some perspective on future research.

5.1 Unit Step, and Congruences

We have mentioned in Section 1 two general techniques for counting the number of integer points in a polyhedron [1, 15]. Their complexity prevents them to be even summarized here. Instead, we give an intuition on why these methods are strictly more powerful than what is presented in this paper.

In a polyhedron description, the domain of a given variable j may be defined with:

$$l(i, \dots) \leq \alpha j < u(i, \dots) \quad (\alpha > 1)$$

where $l(i, \dots)$ and $u(i, \dots)$ are affine functions. Since the congruences of l and u modulo α are variable, the number of integer values of j also depends on these congruences. If j is then involved in additions and summations, the congruences will grow and multiply, some will vanish, and those involving the parameters will eventually remain. This is why, in the general case, the result of counting involves either periodic numbers [1], or integer parts [15]. This informal argument also shows that both techniques capture phenomena that this paper does not even try to model.

One should however keep in mind that the complexity of the general techniques is not strictly mathematical or algorithmic. Here is an example with three non-unit coefficients:

$$0 \leq 2i < N \wedge 0 \leq 3j < N + i \wedge 0 \leq 5k < N + i + j$$

The number of integer points inside this polyhedron, computed with the `barvinok` library [14], is a polynomial of degree 3 (somehow) which is a sum of 46 terms that we have no space to display in full; here are four of these terms:

$$\begin{aligned} & \dots + \frac{1}{2} N^2 \left\lfloor \frac{3+N}{5} \right\rfloor - 2N \left\lfloor \frac{N}{2} \right\rfloor \left\lfloor \frac{3+2N}{5} \right\rfloor \\ & + \frac{3}{2} \left\lfloor \frac{1+N}{3} \right\rfloor^2 \left\lfloor \frac{10+4N}{15} \right\rfloor - \frac{5}{2} \left\lfloor \frac{N}{2} \right\rfloor \left\lfloor \frac{6+3N}{10} \right\rfloor^2 + \dots \end{aligned}$$

We conclude from this example that intersecting polyhedra and lattices, or using non-unit steps in loops, has an intrinsic complexity that applications must be ready to confront.

5.2 Bounds Coherence, and Chambers

One could argue that both Ehrhart and Barvinok-based counting algorithms do not rely on any kind of coherence of bounds or inequalities. The fact is that both methods start by “sanitizing” their input, essentially decomposing it into convex, tightened polyhedra. Both use the same algorithm to do so [9] (see also [3]), whose output is twofold. First, it splits the parameter space into distinct *validity domains* [1], also called *chambers* [15]. Second, it computes for each chamber a different set of vertices, which are the fundamental inputs to both counting algorithm.

The decomposition algorithm produces a set of chambers that are then treated independently, each one getting its own associated counting polynomial. A similar pre-processing step could very well be used to produce different simple

| with n when n >= 0 | (rank) | (count) |
|--------------------|--|-------------------------------|
| do | 0 | $2n + 3n^{2l} + n^{3l}$ |
| for i = 0 to n | 0 | $2n + 3n^{2l} + n^{3l}$ |
| do | $2in - 3i^{2l} + i^{2l}n - 2i^{3l}$ | $2n - 3i + in - 2i^{2l}$ |
| S1 | $2in - 3i^{2l} + i^{2l}n - 2i^{3l}$ | 1 |
| for j = 0 to i | $1 + 2in - 3i^{2l} + i^{2l}n - 2i^{3l}$ | i |
| do S2 done | $1 + 2in - 3i^{2l} + i^{2l}n - 2i^{3l} + j$ | 1 |
| S3 | $1 + i + 2in - 3i^{2l} + i^{2l}n - 2i^{3l}$ | 1 |
| for j = 1+i to n | $2 + i + 2in - 3i^{2l} + i^{2l}n - 2i^{3l}$ | $-2 + 2n - 4i + in - 2i^{2l}$ |
| do | $-3i + 2in - 5i^{2l} + i^{2l}n - 2i^{3l} + 2j + ji$ | 2 + i |
| S4 | $-3i + 2in - 5i^{2l} + i^{2l}n - 2i^{3l} + 2j + ji$ | 1 |
| for k = 0 to i | $1 - 3i + 2in - 5i^{2l} + i^{2l}n - 2i^{3l} + 2j + ji$ | i |
| do S5 done | $1 - 3i + 2in - 5i^{2l} + i^{2l}n - 2i^{3l} + 2j + ji + k$ | 1 |
| S6 | $1 - 2i + 2in - 5i^{2l} + i^{2l}n - 2i^{3l} + 2j + ji$ | 1 |
| done | done | done |

Figure 1. The Cholesky kernel, annotated with counting and ranking polynomials on every program construct. Statement sequences are delimited with do and done, some single instruction sequences are collapsed to avoid repetition.

```
def dyn_unrank (n, RANK):
  i = unisolve ([0, 2*n, -3+n, -2], 0, n, RANK)
  if RANK < 1+2*i*n-3*i*(i-1)//2+i*(i-1)//2*n-2*i*(i-1)*(i-2)//6:
    return ([i], [0, 0])
  elif RANK < 1+i+2*i*n-3*i*(i-1)//2+i*(i-1)//2*n-2*i*(i-1)*(i-2)//6:
    j = unisolve ([1+2*i*n-3*i*(i-1)//2+i*(i-1)//2*n-2*i*(i-1)*(i-2)//6, 1], 0, i, RANK)
    return ([i, j], [0, 1, 0])
  elif RANK < 2+i+2*i*n-3*i*(i-1)//2+i*(i-1)//2*n-2*i*(i-1)*(i-2)//6:
    return ([i], [0, 2])
  else:
    j = unisolve ([-3*i+2*i*n-5*i*(i-1)//2+i*(i-1)//2*n-2*i*(i-1)*(i-2)//6, 2+i], 1+i, n, RANK)
    if RANK < 1-3*i+2*i*n-5*i*(i-1)//2+i*(i-1)//2*n-2*i*(i-1)*(i-2)//6+2*j+j*i:
      return ([i, j], [0, 3, 0])
    elif RANK < 1-2*i+2*i*n-5*i*(i-1)//2+i*(i-1)//2*n-2*i*(i-1)*(i-2)//6+2*j+j*i:
      k = unisolve ([1-3*i+2*i*n-5*i*(i-1)//2+i*(i-1)//2*n-2*i*(i-1)*(i-2)//6+2*j+j*i, 1], 0, i, RANK)
      return ([i, j, k], [0, 3, 1, 0])
    else:
      return ([i, j], [0, 3, 2])
```

Figure 2. Rank inversion code for the Cholesky kernel. It takes the value of n and a rank, and returns the values of the loop counters, as well as an instruction locator. The unisolve function takes a univariate polynomial as a list of coefficients; note that all its calls but the first act on degree-1 polynomials here. For the sake of illustration, the output code is in Python and polynomial expressions have been translated in the most naive way, neither of which we recommend for serious use-cases.

| | | |
|--|---|--|
| <pre>with n m p when n >= 0 and m >= 0 and p >= 0 do for i = 0 to n do for j = 0 to p do for k = 0 to m do S done</pre> | <pre>(rank) 0 0 ip + ipm ip + ipm ip + ipm + j + jm ip + ipm + j + jm 1 + ip + ipm + j + jm 1 + ip + ipm + j + jm + k</pre> | <pre>def dyn_unrank (n, m, p, RANK): i = (RANK - (0)) // (p+p*m) j = (RANK - (i*p+i*p*m)) // (1+m) if RANK < 1+i*p+i*p*m+j+j*m: return ([i, j], [0, 0, 0]) else: k = (RANK - (1+i*p+i*p*m+j+j*m)) // (1) return ([i, j, k], [0, 0, 1, 0])</pre> |
|--|---|--|

Figure 3. Naive matrix multiply, with unranking code where simple arithmetic replaces unisolve() on degree-1 polynomials. The // integer division operator is used here as a floor operation, contrary to Figure 2 where it always perform exact division.

loops, with their associated context. Depending on the use-case, this may take some back-and-forth between loops and polyhedra. But it remains that the bounds coherence validity constraint is anecdotal in such a broader picture.

We take this opportunity to make an out-of-scope, admittedly subjective comment on polyhedral algorithms in general. In hindsight, we feel that the decomposition algorithm from Loechner and Wilde [9] should play a central role in many more polyhedral techniques, because it solves a pervasive problem, and its output is close to a polyhedral normal form. It is our intuition that it could have a tremendous simplifying effect on several major polyhedral tasks.

5.3 Applications

The uses of quantitative aspects (counts) or random access to individual instructions (ranks) are relatively rare in polyhedral optimization techniques. One good example is the load-balanced parallelization of an arbitrary, non-rectangular loop nest [2], as prescribed, for instance, by the OpenMP collapse clause [10]: counting provides the total load, which is divided by the number of available threads; then, unranking lets every thread determine where it must start and end. An extension to this is *algebraic tiling* [13], which follows the same principle but operates along several dimensions.

While counting and ranking is almost always expressed in terms of time (counted in instruction executions), it applies equally well in space (counted in data elements, or bytes). Assimilating loops to arrays and sequences to structures is a quick analogy that enables memory volume quantification and random addressing for any composition of structures and arrays of any affine or even polynomial shape. One step beyond, the work by Clauss and Meister [4] optimizes spatial locality by storing data in the order with which it is accessed. It uses ranking in combining two distinct domains (execution time and memory space).

Because ranking maps a potentially highly structured domain onto a sequential reference dimension, we expect to see more applications using it to find correspondence across such domains. For instance, it could be used to match synchronization primitives across distinct tasks, as in [6], or communication events inside a distributed program, where the sequential dimension is the history of events on the synchronization or communication medium. It is with the aim to better understand such correspondences that we have tried to define a straightforward path from loops to sums to ranking polynomials. We are still far from this goal, and we expect most of the remaining work to focus on polynomials, for which we have introduced a representation that we hope will prove to be more than an amusing algebraic device.

5.4 Implementation

This paper should come with its proof-of-concept implementation. It is written in Python, uses no external library, and covers all aspects of Sections 2 through 4 in less than 600 lines

of code, to which it adds utilities that make it suitable for experimentation. If you read an electronic, PDF version of the paper you can extract source code by clicking on the underlined words in this sentence. Launching this program with no argument gives usage information. About three dozen source loop nests are built-in, including all non-rectangular kernels from the Polybench suite [11], so it also serves as a gallery of examples. Of course, it comes with no guarantee whatsoever, and is certainly not meant for anything but illustrating what you have read here.

References

- [1] Philippe Clauss. 1996. Counting Solutions to Linear and Nonlinear Constraints through Ehrhart Polynomials: Applications to Analyze and Transform Scientific Programs. In *Proceedings of the 10th International Conference on Supercomputing (ICS '96)*. Association for Computing Machinery, 278–285. <https://doi.org/10.1145/237578.237617>
- [2] Philippe Clauss, Ervin Altintas, and Matthieu Kuhn. 2017. Automatic Collapsing of Non-Rectangular Loops. In *Parallel and Distributed Processing Symposium (IPDPS), 2017*. IEEE International, 778 – 787. <https://doi.org/10.1109/IPDPS.2017.34>
- [3] Philippe Clauss and Vincent Loechner. 1998. Parametric Analysis of Polyhedral Iteration Spaces. *Journal of Signal Processing Systems* 19, 2 (July 1998), 179–194. <https://doi.org/10.1023/A:1008069920230>
- [4] Philippe Clauss and Benoît Meister. 2000. Automatic Memory Layout Transformations to Optimize Spatial Locality in Parameterized Loop Nests. *SIGARCH Comput. Archit. News* 28, 1 (mar 2000), 11–19. <https://doi.org/10.1145/346023.346031>
- [5] Paul Feautrier and Christian Lengauer. 2011. *Polyhedron Model*. Springer US, 1581–1592. https://doi.org/10.1007/978-0-387-09766-4_502
- [6] Paul Feautrier, Eric Violard, and Alain Ketterlin. 2014. Improving X10 Program Performances by Clock Removal. In *23rd International Conference on Compiler Construction (CC'14), part of ETAPS'14*. <https://inria.hal.science/hal-00924206>
- [7] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. 1994. *Concrete Mathematics: A Foundation for Computer Science* (2nd ed.). Addison-Wesley Longman Publishing Co., Inc.
- [8] IMPACT Workshop. Since 2011. *Conference Programs*. Retrieved September 1, 2023 from <https://impact-workshop.org/>
- [9] Vincent Loechner and Doran Wilde. 1997. Parameterized Polyhedra and Their Vertices. *International Journal of Parallel Programming* 25 (12 1997). <https://doi.org/10.1023/A:1025117523902>
- [10] OpenMP Architecture Review Board. 2021. OpenMP Application Program Interface. <http://www.openmp.org/specifications>
- [11] Louis Noël Pouchet and Tomofumi Yuki. 2023. *PolyBench/C*. Retrieved September 1, 2023 from <https://sourceforge.net/projects/polybench/>
- [12] William Pugh. 1994. Counting Solutions to Presburger Formulas: How and Why. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*. Association for Computing Machinery, 121–134. <https://doi.org/10.1145/178243.178254>
- [13] Clément Rossetti and Philippe Clauss. 2023. Algebraic Tiling. In *13th International Workshop on Polyhedral Compilation Techniques (IMPACT 2023, in conjunction with HiPEAC 2023)*. <https://impact-workshop.org/impact2023/#rossetti23-algebraic>
- [14] Sven Verdoolaege. 2023. *The barvinok library*. Retrieved September 1, 2023 from <https://barvinok.sourceforge.io/>
- [15] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. 2007. Counting Integer Points in Parametric Polytopes Using Barvinok’s Rational Functions. *Algorithmica* 48 (05 2007), 37–66. <https://doi.org/10.1007/s00453-006-1231-0>