Simplifying Polynomial Constraints Over Integers to Make Dependence Analysis More Precise

Vadim Maslov
vadik@cs.umd.edu

William Pugh
pugh@cs.umd.edu

Institute for Advanced Computer Studies
Dept. of Computer Science
Univ. of Maryland, College Park, MD 20742

Abstract

Why do existing parallelizing compilers and environments fail to parallelize many realistic FORTRAN programs? One of the reasons is that these programs contain a number of linearized array references, such as $A(M*N*i*N*j*k)$ or $A(i*(i+1)/2+j)$. Performing exact dependence analysis for these references requires testing polynomial constraints for integer solutions. Most existing dependence analysis systems, however, restrict themselves to solving affine constraints only, so they have to make worst-case assumptions whenever they encounter a polynomial constraint.

In this paper we introduce an algorithm which exactly and efficiently solves a class of polynomial constraints which arise in dependence testing. Another important application of our algorithm is to generate code for loop transformation known as symbolic blocking (tiling).

This work is supported by an NSF PYI grant CCR-9157384 and by a Packard Fellowship.
Simplifying Polynomial Constraints Over Integers
to Make Dependence Analysis More Precise

Vadim Maslov    William Pugh
vadik@cs.umd.edu, 301-405-2726  pugh@cs.umd.edu, 301-405-2705
Dept. of Computer Science
Univ. of Maryland, College Park, MD 20742

February 26, 1994

Abstract

Why do existing parallelizing compilers and environments fail to parallelize many realistic FORTRAN programs? One of the reasons is that these programs contain a number of linearized array references, such as \( A(M*N*i+M*j+k) \) or \( A(i*(i+1)/2+j) \). Performing exact dependence analysis for these references requires testing polynomial constraints for integer solutions. Most existing dependence analysis systems, however, restrict themselves to solving affine constraints only, so they have to make worst-case assumptions whenever they encounter a polynomial constraint.

In this paper we introduce an algorithm which exactly and efficiently solves a class of polynomial constraints which arise in dependence testing. Another important application of our algorithm is to generate code for loop transformation known as symbolic blocking (tiling).

KEYWORDS: parallelizing compilers, polynomial constraints, linearized subscripts, symbolic blocking.

1 Introduction

In this paper we describe techniques for simplifying polynomial constraints. This work supersedes our previous work on dependence testing of non-linear subscripts [Mas92] and allows us to handle polynomial constraints that arise in a number of situations. This work is also an extension of the Omega test [Pug92, PW92, PW93] — the system that simplifies conjunctions of affine constraints over integers and performs exact elimination of existentially quantified variables.

Polynomial constraints arise in a number of situations:

- Dependence between references with linearized subscripts,
- Complicated terms in subscript functions arising from induction variable recognition,
- Tiling with symbolic block sizes (i.e., block sizes specified by a variable and not a constant).

Let’s consider some examples of problems with polynomial constraints. All the problems are taken from the real-life programs and simplifying them exactly is crucial for our ability to parallelize these programs.

\[
\begin{align*}
&\text{do } \text{i=p}, \text{p+L-1} \\
&\quad \text{do } \text{j=q}, \text{q+N-1} \\
&\quad \quad \text{do } \text{k=r}, \text{r+M-1} \\
&\quad \quad \quad A(M*N*i+N*j+k) = \ldots \\
&\quad \quad \text{S1: } A(M*N*i+N*j+k) = \ldots \\
&\quad \text{S2: } \ldots = A(M*N*i+N*j+k) \\
&\quad \text{endo} \\
&\quad \text{endo} \\
&\quad \text{endo} \\
\end{align*}
\]

\[MN_i e + N_j e + k e = MN_i r + N_j r + k r \]
\[p \leq i e , i r \leq p + L - 1 \]
\[q \leq j e , j r \leq q + M - 1 \]
\[r \leq k e , k r \leq r + N - 1 \]

(1)

\[MN_i e + N_j e + k e = MN_i r + N_j r + k r \]
\[p \leq i e , i r \leq p + L - 1 \]
\[q \leq j e , j r \leq q + M - 1 \]
\[r \leq k e , k r \leq r + N - 1 \]

(2)

\[
\begin{align*}
&\text{do } \text{i=p}, \text{p+L-1} \\
&\quad \text{do } \text{j=q}, \text{q+N-1} \\
&\quad \quad \text{do } \text{k=r}, \text{r+M-1} \\
&\quad \quad \quad A(M*N*i+N*j+k) = \ldots \\
&\quad \quad \text{S1: } A(M*N*i+N*j+k) = \ldots \\
&\quad \text{S2: } \ldots = A(M*N*i+N*j+k) \\
&\quad \text{endo} \\
&\quad \text{endo} \\
&\quad \text{endo} \\
\end{align*}
\]

Figure 1: Product of variable and symbolic constant(s)
The program in Figure 1 is one of many loop nests from the oil reservoir simulation program BOAST in the RiCEPS benchmark suite. This is rather typical example of loop nest with linearized references, which are met quite often in real programs, and [Mas92] discusses in length why linearization is used.

To be able to parallelize the \( i, j \) and \( k \) loops, we need to prove that the flow dependence from the statement instance \( S_1[i_w, j_w, k_w] \) to the statement instance \( S_2[i_r, j_r, k_r] \) is loop-independent. This dependence is described by the set of constraints (1) that we want to be able to simplify to (2).

All existing dependence analysis techniques (that we know of) except for one fail to prove that this dependence is loop-independent. *Symbolic delinearization* [Mas92] can prove this, but it has serious limitations discussed in Section 8.

### 1.2 Triangular linearization

Consider the program in Figure 2. Since the one-dimensional array \( A \) is a linearized version of a triangular matrix \( A \), a reference to \( A(i, j) \) is expressed as \( A(i^*(i+1)/2 + j) \). Linearized triangular matrices are used quite often in scientific codes.

Loop \( i \) cannot be parallelized unless we know that flow dependence from \( S_1[i_w, j_w] \) to \( S_2[i_r, j_r] \) is loop-independent, that is, \( i_w = i_r \). No existing dependence test (that we know of) can automatically prove this. The problem describing this dependence is (3) and our techniques simplify it to (4), which proves that the dependence is loop-independent. Since we also know that \( j_w = j_r \), we can fuse the two \( j \) loops if we need to, and since existing techniques cannot establish that \( j_w = j_r \), they cannot perform fusion for this example.

### 1.3 Code generation for symbolic blocking (tiling)

Consider the program fragment in the left column of Figure 3. To improve locality (that is, to better use the memory cache) the loop transformation known as *blocking* or *tiling* [AK87] is applied to this loop nest. The idea of transformation is to block (tile) the iteration space into blocks such that each block requires data which fits into cache. We perform loop blocking using a framework of *Uniform Loop Transformations*
The mapping from the old iteration space to the new iteration space is described by a relation

\[ S_i : [i_o, j_o] \rightarrow [i_b, j_n, i_n] \quad | \quad i_b = [(i_o - 1)/N] + 1 \land j_n = j_o \land i_n = i_o \]  

(7)

where \(i_o, j_o\) are old loop variables and \(i_b, j_n, i_n\) are new loop variables.

If the blocking factor \(N\) is a known integer constant, then we have a pseudo-affine schedule and blocked code is generated without difficulty. However, if \(N\) is a symbolic constant, then our schedule becomes non-affine. So to generate the blocked code shown in the right column of Figure 3, we (among other things) have to compute loop parameters for the new loop \(i_B\). The constraints on \(i_B\) are described by the problem (5). Our algorithm simplifies it to the problem (6) and then we extract the initial value of \(i_B\) and upper bound of \(i_B\).

2 Our approach

Our basic approach is to try to transform a general polynomial constraint into a conjunction of affine constraints and one of the special forms that we later affinize:

\[
\begin{align*}
xy &\geq c & \text{A hyperbolic inequality} \\
x &\geq 1 & \\
y &\geq 1 & \\
a_x x^2 + a_y y^2 &\geq c & \text{An elliptical inequality} \\
a_x x^2 + a_y y^2 &= c & \text{An elliptical equality}
\end{align*}
\]

(8)

Here \(x\) and \(y\) are variables and \(a_x, a_y\) and \(c\) are constants. These transformations are mostly factoring and they are discussed in detail in Section 4.

Then we affinize the special form constraints. For example, \(xy \geq 5\) is equivalent to (the first conjunction of this is shown in Figure 4):

\[
(x \geq 1 \land 2x + y \geq 7 \land x + y \geq 5 \land x + 2y \geq 7 \land y \geq 1) \lor (x \leq -1 \land 2x + y \leq -7 \land x + y \leq -5 \land x + 2y \leq -7 \land y \leq -1)
\]

This is, perhaps, one of the few places where the fact that we are working with integers makes things easier than if we were working with reals (it is not possible to convert polynomial constraints over real variables into affine form). Details of affinization are given in Section 6.

Of course, not all polynomial constraints are of the form that we can factor and affinize. The details of the algorithm that systematically applies factoring and affinization are given in Section 3.
Problem SimplifyPolynomial(Problem p) Begin
   Boolean change := True
   Integer n := 0
   \( p_n = p \)
   Do while (problem \( p_n \) has polynomial constraints \( \land change \))
      change := False
      \( v := \) affine variables of \( p_n \)
      \( p_{n+1} := \exists v \ s.t. \ p_n \)
      \( q_{n+1} := \) gist \( p_{n+1} \) given \( p_n \)
      \( n := n + 1 \)
      If we can determine that \( p_n \) is unsatisfiable then Return(False)
      For each constraint \( c \) in polynomial constraints of \( p_n \)
         Try to factor and affinize constraint \( c \)
         If (factoring and/or affinization succeeds) \( change := True \)
   EndFor
   EndDo
   \( p = p_n \land q_n \land q_{n-1} \land \cdots \land q_1 \)
   If polynomial constraints remain in \( p \)
      Use affine equalities in \( p \) to derive substitutions
      Try to simplify or eliminate polynomial constraints using substitutions
      If this produces additional affine equalities, repeat
   EndIf
   Return(p)
End

Figure 5: Polynomial constraints simplification algorithm

3 Algorithm to simplify polynomial constraints

In this section we present our top-level algorithm to simplify the conjunction of a set of polynomial and affine constraints.

Representing polynomial constraints. We use the following extension of the Omega test framework to represent polynomial constraints. For each product of regular variables that we encounter in a polynomial constraint we create a product variable that represents it. Then we divide polynomial problem in two parts:

- **Affine part** that is original problem in which products were replaced with product variables,
- **Product part** that essentially is a definition of product variables in terms of regular variables.

For example, polynomial problem

\[
N j_w + k_w = N j_r + k_r \land 1 \leq j_w N - N j_r + N \land q \leq j_w, j_r \leq q + M - 1
\]

is represented as (product part uses := for defining product variables):

\[
v_1 + k_w = v_2 + k_r \land 1 \leq v_1 - v_2 + N \land q \leq j_w, j_r \leq q + M - 1
\]

\[
v_1 := N j_w, \quad v_2 := N j_r
\]

Please note, that products \( N j_w \) and \( j_w N \) are the same and therefore are referred to as one variable \( v_1 \).

We further classify regular variables as:

- **Affine variables.** These are variables that do not appear in products. We single them out because when we can eliminate affine variables using the Omega test without losing precision.
- **Semi-affine variables.** These are regular variables that appear in products. We cannot project them out using the Omega test, because they are involved in polynomial constraints.

In the above example affine variables are \( k_w, k_r, q, M \), semi-affine variables are \( j_w, j_r, N \) and product variables are \( v_1, v_2 \).
We can simplify the affine part of a polynomial problem using the Omega test. However, when it comes to factoring and affinization, we use definitions of the product variables from the product part of the problem. Product variables that become unused, are removed.

**Algorithm itself.** We present the algorithm that simplifies a polynomial problem in Figure 5. The algorithm applies factoring and affinization as many times as it can. Each affinization lowers the order of polynomial constraint by 1. So finally we either get affine problem or stop because no affinization nor factoring can be done. Therefore algorithm always terminates.

To satisfy conditions for factoring and affinization we eliminate affine variables that stand in the way of factoring. Basically our goal is to get polynomial constraint that has less variables than original constraint, to factor out the common term (or apply more intricate factoring, as in triangular delinearization example) and to affinize it.

Variables that are removed as a result of projecting out affine variables and constraints involving these variables are memorized in $q_i$ problems. When simplification is finished, we use $q_i$ problems to restore the original problem. As restoration goes on, we use new equalities and inequalities produced by affinization to simplify restored polynomial constraints.

## 4 Factoring

We use several techniques to transform a polynomial constraint to one of the forms (8). These techniques are described for inequality constraints, but they work equally well for equalities.

**Common term.** If a factor $x$ occurs in all terms of a constraint, except for a constant term, we can factor this constraint. That is, we transform the constraint

$$
\sum_{i=1}^{n} a_i x R_i \geq c
$$

where $a_i$ and $c$ are integer constants, $x$ is a variable, each $R_i$ is a product of variables or the constant 1 to

$$
\exists \ y \ s.t. \ \ y = \sum_{i=1}^{n} a_i R_i
$$

So we reduce the order of the original polynomial constraint by 1, hopefully making it affine, and we produce a hyperbolic constraint that can be affinized.

**Breaking quadratic constraint.** As a more specialized case, a constraint of the form:

$$
a_x^2 x^2 + b_x x - a_y^2 y^2 - b_y y + c \geq 0
$$

where $a_x > 0, a_y > 0, b_x, b_y$ and $c$ are known integer constants, $x$ and $y$ are variables, is transformed to the following equivalent constraint (that involves hyperbolic equality or inequality):

$$
\exists \ a, \beta \ s.t. \quad \beta = 2a_x^2 a_y x + 2a_x a_y^2 y + b_x a_y + b_y a_x
$$

If the coefficient of $x^2$ (that is, $a_x^2$) is not the square of some integer, we should multiply the whole constraint by a positive integer constant which makes the coefficient of $x$ a square. If after this the coefficient of $y^2$ (that is, $a_y^2$) is not a square, factoring cannot be done in integers, and therefore we give up on this constraint.
Completing square. A constraint of the form:

\[ a_x x^2 + b_x x + a_y y^2 + b_y y + c \geq 0 \]

where \( a_x > 0, a_y > 0, b_x, b_y \) and \( c \) are known integer constants, \( x \) and \( y \) are variables, is transformed to the following equivalent set of constraints (involving elliptical equality or inequality):

\[
\exists \alpha, \beta \text{ s.t. } \begin{align*}
\alpha &= 2a_x x + b_x \\
\beta &= 2a_y y + b_y \\
\alpha^2 + \alpha \beta^2 &\geq \frac{\alpha b_x^2 + \alpha b_y^2}{-4a_x a_y c}
\end{align*}
\]

5 Representing integer division.

We transform constraint \( L \) that involves integer division (here \( E \) and \( F \) are affine expressions):

\[ L([E/F], \ldots) \]

to polynomial constraint:

\[ \exists t, \alpha \text{ s.t. } L(t, \ldots) \land tF + \alpha = E \land 0 \leq \alpha \leq F - 1 \]

6 Affinization

Let’s consider the area described by a constraint (we discuss only \( \leq \)-inequalities here; inequalities with \( <, >, \) or \( \geq \) operators can be converted into \( \leq \)-inequalities):

\[ x_l \leq x \leq x_r \land y \leq f(x) \]

We require this area to be \textit{convex}, that is, \( \forall x: x_l \leq x \leq x_r \Rightarrow f''(x) \leq 0 \). If it is not so, we can break the segment \([x_l, x_r] \) into several segments, such that this requirement is satisfied in each segment, and consider these segments separately. When we have the convex area, the derivative of \( f(x) \) steadily decreases as \( x \) increases, so we can break interval \([x_l, x_r] \) into 4 intervals (some of them can be empty):

\[
\begin{align*}
x_l \leq x \leq x_1 &\Rightarrow 1 \leq f'(x) \\
x_1 \leq x \leq x_0 &\Rightarrow 0 \leq f'(x) < 1 \\
x_0 \leq x \leq x_{-1} &\Rightarrow -1 \leq f'(x) < 0 \\
x_{-1} \leq x \leq x_r &\Rightarrow f'(x) \leq -1
\end{align*}
\]

Affinization Theorem. If the above conditions are satisfied, the non-affine constraint (9) is equivalent to a conjunction of affine constraints:

\[
[x_l] \leq x \leq [x_r] \land y \leq [f(x_l)] \land \\
\bigwedge_{i=x_l}^{x_r-1} \text{line } ((i, [f(i)]), (i+1, [f(i+1)])) \leq 0 \land \\
\bigwedge_{i=[f(x_l)]}^{[f(x_r)]} \text{line } (([f^{-1}(i)], i), ([f^{-1}(i+1)], i+1)) \leq 0 \land \\
\bigwedge_{i=[f(x_l)]}^{x_{-1}} \text{line } (([f^{-1}(i)], i), ([f^{-1}(i+1)], i+1)) \geq 0 \land \\
\bigwedge_{i=x_{-1}}^{x_r} \text{line } ((i-1, [f(i-1)]), (i, [f(i)])) \leq 0
\]

(10)

Here the function \( f^{-1}_+(y) \) is inverse function of \( f(x) \) for \( x_1 \leq x \leq x_0 \) and \( f^{-1}_-(y) \) is inverse function of \( f(x) \) for \( x_0 \leq x \leq x_{-1} \). The function \( \text{line } ((x_1, y_1), (x_2, y_2)) \) gives back an expression that is zero along the straight line passing through the points \( (x_1, y_1) \) and \( (x_2, y_2) \), positive to the left of that line (as we move from \( (x_1, y_1) \) to \( (x_2, y_2) \)) and negative to the right of that line:

\[ \text{line } ((x_1, y_1), (x_2, y_2)) = (x_2 - x_1)(y - y_1) - (y_2 - y_1)(x - x_1) \]

In the rest of this section we show how the affinization theorem is applied to hyperbolic and elliptical inequalities.
6.1 Affinizing inequalities

Hyperbolic inequalities with positive constant. To affinize the inequality
\[ xy \geq c \]
where \( c \geq 1 \), we break the domain of \( f(x) = c/x \) into 2 convex areas: \( xy \geq c \geq 1 \equiv (x \leq -1 \land y \leq c/x) \lor (x \geq 1 \land y \geq c/x) \). Applying the affinization theorem to each area, we get that the inequality \( xy \geq c \geq 1 \) is equivalent to:
\[
\begin{align*}
&x \geq 1 \land y \geq 1 \land \bigwedge_{i=1}^{\lceil \sqrt{c-1} \rceil} \left( \text{line} ((i, \left\lfloor \frac{c}{i} \right\rfloor), (i + 1, \left\lfloor \frac{c}{i+1} \right\rfloor)) \geq 0 \land \text{line} ((\left\lfloor \frac{c}{i+1} \right\rfloor, i), (\left\lfloor \frac{c}{i} \right\rfloor, i+1)) \leq 0 \right) \lor \\
&x \leq -1 \land y \leq -1 \land \bigwedge_{i=1}^{1-\sqrt{c}} \left( \text{line} ((i, \left\lceil \frac{c}{i} \right\rceil), (i - 1, \left\lceil \frac{c}{i-1} \right\rceil)) \geq 0 \land \text{line} ((\left\lceil \frac{c}{i-1} \right\rceil, i), (\left\lceil \frac{c}{i} \right\rceil, i-1)) \leq 0 \right)
\end{align*}
\]

Hyperbolic inequalities with non-positive constant. The inequality \( xy \geq 0 \) is equivalent to \((x \geq 0 \land y \geq 0) \lor (x \leq 0 \land y \leq 0)\).

If \( c \leq -1 \), then the inequality \( xy \geq c \) describes non-convex area between two hyperbola branches. We transform this inequality to negation of positive-constant hyperbolic inequality:
\[ xy \geq c \equiv -(x'y' \geq c') \]
where \( x' = -x, y' = y \), \( c' = 1 - c \geq 1 \)

Negation of conjunct produces disjunction of several constraints.

Elliptical inequalities. Affinizing elliptical inequalities is similar to affinizing hyperbolic inequalities.

6.2 Affinization of equalities

An equality constraint of the form \( xy = c \) or \( a_xx^2 + a_yy^2 = c \) has a finite number of integer solutions. We convert this constraint to affine form by enumerating these solutions.

Hyperbolic equalities. Let’s consider the hyperbolic equality \( xy = c \). We replace it with a disjunction of several constraints. For each \( t \) up to \( \lfloor \sqrt{|c|} \rfloor \), if \( t \) divides \( c \) then we add to the disjunction constraints:
\[
\begin{align*}
&x = t \quad \land \quad y = c/t \quad \lor \\
&x = -t \quad \land \quad y = -c/t
\end{align*}
\]

For example, the equality \( xy = 5 \) is equivalent to: \((x = 1 \land y = 5) \lor (x = 5 \land y = 1) \lor (x = -1 \land y = -5) \lor (x = -5 \land y = -1)\).

Elliptical equalities. Similar to hyperbolic equalities.

6.3 Number of constraints generated

For hyperbolic inequalities and equalities number of constraints generated is \( O(\sqrt{c}) \) where \( c \) is constant from (8). As our preliminary study shows, \( c \) values are usually small, and that means that few constraints need to be generated.

Affinizing polynomial constraint only over the feasible domain. Often, we can further restrict number of constraints generated if we know the range of the participating variables is limited. Before generating affine constraints, we find the rectangular bounding box for \( x \) and \( y \) (the Omega test has this capability):
\[
I_x = \min x, \quad I_y = \min y, \quad U_x = \max x, \quad U_y = \max y
\]
Then constraints that do not intersect with the bounding box are not generated at all.

For example, if we know that $x$ is non-negative, then $xy \geq 2$ is equivalent to affine set of constraints $x \geq 1 \land x + y \geq 3 \land y \geq 1$.

7 Examples

7.1 Rectangular delinearization

We start with computing $p_1$, the projection of (1) onto variables involved in products ($i_w, i_r, j_w, j_r, M$ and $N$), and $q_1$, everything else:

$$p_1 \equiv 1 - N \leq MNi_w + Nj_w - MNi_r - Nj_r \leq N - 1 \quad \Rightarrow \quad 1 - M \leq j_w - j_r \leq M - 1$$

$$q_1 \equiv p \leq i_w, i_r \leq p + L - 1, \quad q \leq j_w, j_r \leq q + M - 1 \quad r \leq k_w, k_r \leq r + N - 1$$

We can factor and affinize the polynomial constraints. The constraints in $p_1$ imply $N \geq 1$, so we generate only one branch of hyperbola:

$$1 \leq N(Mi_w - Mi_r + j_w - j_r + 1)$$

$$1 - M \leq j_w - j_r \leq M - 1$$

$$1 \leq Mi_w - Mi_r + j_w - j_r + 1 \quad \Rightarrow \quad 1 \leq Mi_w, j_w \leq M - 1$$

We again eliminate all affine variables ($N, j_w, j_r$):

$$p_2 \equiv 1 - M \leq Mi_r - Mi_w \leq M - 1 \quad \Rightarrow \quad 1 \leq M(i_w - i_r + 1)$$

$$q_2 \equiv Mi_w + j_w = Mi_r + j_r$$

Affinizing we get $p_2 \equiv 1 \leq M \land i_w = i_r$.

Having reduced $p$ to affine form, we restore constraints involving eliminated variables and simplify:

$$p \equiv p_2 \land q_2 \land q_1 \equiv \frac{i_w}{i_r} = i_r, \quad MNi_w + Nj_w + k_w = MNi_r + Nj_r + k_r$$

Before applying our algorithm to the problem (3), we convert integer division by 2 to integer multiplication:

$$p \equiv \exists t_1, t_2, \alpha, \beta s.t.$$

$$2t_1 + \alpha = i_r + i_1 \land 0 \leq \alpha \leq 1 \quad 0 \leq j_2 \leq N$$

$$2t_2 + \beta = i_2 + i_2 \land 0 \leq \beta \leq 1 \quad 0 \leq \beta \leq i_1 \leq N$$

$$t_1 + j_1 = t_2 + j_2 \land 0 \leq j_1, i_2 \leq N - 1 \quad \Rightarrow \quad i_2 + 2j_2 + i_2 = i_1 + 2j_1 + i_2$$

$$0 \leq j_1 \leq i_1 \land 0 \leq j_2 \leq i_2$$

We now compute $p_1$, the projection of $p$ onto variables involved in products, and $q_1$, everything else needed so that $p = p_1 \land q_1$:
We factor the polynomial constraints in \( p_1 \) and affine the factored forms:

\[
p_1 \equiv \begin{cases} 
0 \leq i_1 \\
0 \leq i_2 \\
i_2 + i_1^2 \leq 1 + 3i_1 + i_1^2 \\
i_1 + i_1^2 \leq 1 + 3i_1 + i_1^2 
\end{cases}
q_1 \equiv \begin{cases} 
0 \leq j_2 \leq i_5 < N \\
0 \leq j_1 \leq i_5 < N \\
i_2 + 2j_2 + i_1^2 \leq 1 + i_1 + 2j_1 + i_1^2 \\
i_1 + 2j_1 + i_1^2 \leq 1 + i_1 + 2j_1 + i_1^2 
\end{cases}
\]

We factor the polynomial constraints in \( p_1 \):

\[
i_1^2 + 3i_1 - i_2 \geq 0 \quad \Rightarrow \quad (i_1 - i_2 + i_1 + i_1 + 2) \geq 2
\]

and affline the factored forms:

\[
(i_1 - i_2 + 1)(i_1 + i_2 + 2) \geq 2 \quad \Rightarrow \quad i_1 - i_2 + 1 \geq 1 \land i_1 + i_2 + 2 \geq 1 \land (i_1 - i_2 + 1) + (i_1 + i_2 + 2) \geq 3
\]

Replacing these two polynomial constraints with their affine equivalent and simplifying yields:

\[
p_1 \equiv 0 \leq i_1 = i_2
\]

Since \( p_1 \) is completely affine, we are done. We combine \( p_1 \) and \( q_1 \) and simplify, yielding:

\[
p \equiv p_1 \land q_1 \equiv \begin{cases} 
i_1 = i_2 \\
0 \leq j_2 \leq i_5 < N \\
0 \leq j_1 \leq i_5 < N \\
i_2 + 2j_2 + i_1^2 \leq 1 + i_1 + 2j_1 + i_1^2 \\
i_1 + 2j_1 + i_1^2 \leq 1 + i_1 + 2j_1 + i_1^2
\end{cases}
\]

By substituting \( i_1 \) for \( i_2 \), we get:

\[
\begin{align*}
p &\equiv \begin{cases} 
i_2 := i_1 \\
i_3 := i_1 \\
i_5 := i_1 \\
0 \leq j_2 \leq i_5 < N \\
0 \leq j_1 \leq i_5 < N \\
0 \leq j_2 \leq i_5 < N \\
0 \leq j_1 \leq i_5 < N \end{cases} \\
i_2 + 2j_2 + i_1^2 \leq 1 + i_1 + 2j_1 + i_1^2 \\
i_1 + 2j_1 + i_1^2 \leq 1 + i_1 + 2j_1 + i_1^2 \\
2j_2 \leq 1 + 2j_1 \\
j_2 \leq j_1 \\
j_2 = j_1 \\
j_2 = j_1 \\
j_2 = j_1 \end{align*}
\]

This is the final result (4).

### 7.3 Symbolic blocking

Using our algorithm we can simplify polynomial problems that arise in code generation for symbolic blocking. Let’s consider code generation for program in Figure 3. The constraints on old loop variables are:

\[
IS = 1 \leq j_5 \leq i_6 \leq M
\]

We use them and schedule (7) to compute constraints on new loop variables. First, we convert the equality

\[
i_b = \lfloor (i_6 - 1)/N \rfloor N + 1
\]

used in the schedule to polynomial constraint

\[
\exists t \ s.t. \ i_b = tN + 1 \land tN + a = i_6 - 1 \land 0 \leq a \leq N - 1
\]

So constraints on new variables \( i_b, j_5, i_n \) are:

\[
1 \leq j_5 \leq i_6 \leq M \\
i_b = tN + 1
\]

\[
IS' = \exists i_b, j_5, t, a \ s.t. \\
\begin{align*}
tN + a &= i_6 - 1 \\
0 &\leq a \leq N - 1 \\
j_5 &= j_5 \land i_n = i_6
\end{align*}
\]

We use the problem \( IS' \) to generate new loop bounds, proceeding from the outer loop \( i_b \) to the inner loop \( i_n \):
To compute bounds on \(i_b\) we should eliminate from \(IS'\) all affine variables except \(i_b\) and symbolic constants. First, we use our polynomial algorithm to simplify \(IS'\). Eliminating all affine variables and affinizing produces:

\[
p_1 \equiv N + tN \geq 1 \land N \geq 1 \equiv N(t + 1) \geq 1 \land N \geq 1 \equiv t \geq 0 \land N \geq 1
\]

Adding \(p_1\) to the original problem and simplifying produces:

\[
IS' = \exists t \ s.t. \begin{cases} 1 \leq j_n \leq i_n \leq M \\ i_b = tN + 1 \\ tN < i_n \leq N + tN \\ t \geq 0 \end{cases}
\]

Projecting \(IS'\) onto \(i_b\) and symbolic constants produces:

\[
p_{i_b} \equiv \exists t \ s.t. \begin{cases} i_b = tN + 1 \\ 2 - N \leq i_b \leq M \\ M \geq 1 \land N \geq 1 \\ t \geq 0 \end{cases}
\]  

(11)

Having these bounds, code generation routine determines that \(i_b\) is step-\(N\) variable and that the step is positive. Now we should extract an initial value of \(i_b\) from these constraints. The initial value of \(i_b\) is the lowest possible number that satisfies \(IS'\). As long as we have polynomial equality, we can not be sure that the inequality \(expr < i_b\) means that \(expr\) is the initial value, because it’s possible that \(expr\) does not satisfy the polynomial equality (in this example \(2 - N\) is not divided by \(N\), so \(2 - N\) is not the initial value).

To circumvent this difficulty, we instead compute the lower bound on the iteration counter \(t\). We project out variable \(i_b\) and affinize \(p_{i_b}\) to get constraints on \(t\):

\[
\exists i_b \ s.t. \begin{cases} i_b = tN + 1 \\ 2 - N \leq i_b \leq M \\ M \geq 1 \land N \geq 1 \\ t \geq 0 \end{cases} \equiv \begin{cases} 2 - N \leq N + 1 \leq M \\ M \geq 1 \land N \geq 1 \end{cases} \equiv \begin{cases} 0 \leq t \\ t \geq 0 \end{cases}
\]  

(12)

Here we have no equality constraints, so we are sure that the initial value of \(t\) is 0, and therefore the initial value of \(i_b\) is \(t \ast 0 + 1 = 1\).

The \(iB\) loop second parameter ("final" value of \(i_b\)) does not have to be tight, so we can use the inequality \(i_b \leq M\) to decide that \(M\) is the "final" value of \(i_b\).

- Bounds on \(j_n\) are: gist \(\pi_{M,N,i_b,j_n,t,N}(IS')\) given (11). Simplifying we get:

\[
1 \leq j_n \leq M \land j_n \leq N + i_b - 1
\]  

(13)

- Bounds on \(i_n\) are gist \(\pi_{M,N,i_b,j_n,i_n,t,N}(IS')\) given (11) \(\land\) (13). Simplification produces:

\[
j_n \leq i_n \leq M \land i_b \leq i_n \leq N + i_b - 1
\]  

(14)

Having constraints (12) and (13) and (14), uniform methods loop generation routine generates loops as shown in the right column of the Figure 3.

8 Related Work

**Polynomial constraints simplification vs delinearization.** In this paragraph we compare our polynomial constraints simplification algorithm with symbolic delinearization [Mas92].
First, we prove that our algorithm exactly simplifies all problems that can be handled by symbolic
delinearization. The essence of delinearization is transforming the constraints:
\[ N \epsilon_1 + \epsilon_2 = 0, \quad 1 - N \leq \epsilon_2 \leq N - 1 \]
to
\[ \epsilon_1 = 0, \quad \epsilon_2 = 0 \]
where \( \epsilon_1, \epsilon_2 \) are expressions, and \( N \) is a variable. Using our algorithm, we substitute \( \epsilon_2 = -N \epsilon_1 \) into the
inequality \( 1 - N \leq \epsilon_2 \leq N - 1 \). Factoring and simplifying produces:
\[ N(1 - \epsilon_1) \geq 1 \land N(1 + \epsilon_1) \geq 1. \]
Affining both inequalities we get \( \epsilon_1 \geq 0 \land \epsilon_1 \leq 0 \) and therefore \( \epsilon_1 = 0 \). Substituting this equality to the
original problem, we finally prove that \( \epsilon_1 = \epsilon_2 = 0 \).

Symbolic delinearization has several serious restrictions that are not present in our algorithm:
- It can handle only subscript functions linearized according to FORTRAN rules: reference \( A(i_1, i_2, ..., i_n) \)
to array \( A(0 : D_1, 0 : D_2, ..., 0 : D_n) \) is converted to \( A(i_1 + D_1 i_2 + \cdots + D_1 \cdots D_{n-1} i_n) \). We call this
rectangular linearization. Triangular linearization (see Section 1.2) that is used quite often in scientific
codes is not handled.
- Even for the case of rectangular linearization it cannot handle constraints imposed by triangular itera-
tion space.

**Paraphrase-2.** In [HP91] the authors propose to use a symbolic version of Banerjee’s inequalities for de-
pendence testing, but it is known that Banerjee’s inequalities do not detect independence in case of linearized
subscript functions [Mas92].

To alleviate the inexactness of Banerjee’s inequalities, Haghighat and Polychronopoulos propose to detect
monotonically increasing and decreasing subscript function using the finite differences method [HP93]. When
the subscript function is monotonically changing, the reference cannot hit the same memory cell on the next
iteration, and therefore no output dependence can exist from the reference to itself. Our induction variable
recognition system also can discover that the closed form of induction variable is monotonically changing
and we are able to use this fact to prove the absence of the output dependence.

However, when monotonicity cannot be proven — it happens, for example, for program in Figure 2 —
Haghighat and Polychronopoulos finite difference method cannot be used and their techniques cannot prove
that the flow dependence in this example is loop-independent and output dependence does not exist.

**Other approaches.** A number of computer algebra books and papers [KL92, Buc85, DST88] are devoted
to solving polynomial constraints over the complex and real numbers. Since we are interested in polynomial
constraints over the integers, we cannot directly use their results. However, we should study adaptation of
their factoring techniques to our needs.

**9 Conclusion**

We have presented an algorithm that exactly simplifies conjunctions of affine and polynomial constraints
over integers.

This algorithm is not capable of affining arbitrary polynomial constraint. However, we do not always
need complete affinization. For example, in the symbolic blocking example we produce simplified polynomial
problems that can be directly used for generating loop bounds for the loop \( \textbf{iB} \).

Our algorithm for simplifying polynomial constraints is expandable: we can add more sophisticated
factoring techniques to it, and we can consider affinization of more complicated constraints than 2-variable
hyperbolic and elliptical inequalities and equalities (8). We think that expansion of the algorithm should be
guided by practical needs of the concrete application.

More experiments are needed to prove that this set of techniques can affinize most of the polynomial
problems that arise in dependence testing and other parallelizing compiler analyses. Preliminary investigation
shows that we can affinize polynomial constraints that arise due to rectangular and triangular linearization
of subscript functions which seems to be a primary source of polynomial constraints.
References


