UMIACS-TR-93-68.1 February, 1994
CS-TR-3109.1

Simplifying Polynomial Constraints Over Integers to Make Dependence Analysis More Precise

Vadim Maslov
vadik@cs.umd.edu
Dept. of Computer Science

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 — vadik@cs.umd.edu, 301-405-2726
William Pugh — 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+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).

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.

```
    do i=p,p+L-1
      do j=q,q+M-1
        do k=r,r+N-1
S1:       A(M*N*i+N*j+k) = ...
S2:       ... = A(M*N*i+N*j+k)
        enddo
      enddo
    enddo
```

$$\begin{array}{l} MNi_w + Nj_w + k_w = MNi_r + Nj_r + k_r \\ p \le i_w, i_r \le p + L - 1 \\ q \le j_w, j_r \le q + M - 1 \\ r \le k_w, k_r \le r + N - 1 \end{array} \tag{1}$$

$$\begin{array}{l} p \le i_w = i_r \le p + L - 1 \\ q \le j_w = j_r \le q + M - 1 \\ r \le k_w = k_r \le r + N - 1 \\ M \ge 1 \ \wedge \ N \ge 1 \end{array} \tag{2}$$

Figure 1: Product of variable and symbolic constant(s)

```
   do i=0,N-1
     do j=0,i
S1:    A(i*(i+1)/2 + j) = ...
     enddo
     do j=0,i
S2:    ... = A(i*(i+1)/2 + j)
     enddo
   enddo
```

$$\begin{array}{rcl} \frac{i_w(i_w+1)}{2} + j_w & = & \frac{i_r(i_r+1)}{2} + j_r \\ 0 & \le i_w, i_r \le & N-1 \\ 0 & \le j_w \le & i_w \\ 0 & \le j_r \le & i_r \end{array} \tag{3}$$

$$0 \le j_w = j_r \le i_w = i_r \le N-1 \tag{4}$$

Figure 2: Product of two variables: triangular linearization

```
   do i=1,M
     do j=1,i
S1:    A(i,j) = A(j,i)
     enddo
   enddo
```

$$\exists i_o, j_o, i_n, j_n, \; t, \alpha \; s.t. \quad \begin{array}{c} 1 \le j_o \le i_o \le M \\ i_b = tN + 1 \\ tN + \alpha = i_o - 1 \\ 0 \le \alpha \le N-1 \\ j_n = j_o \wedge i_n = i_o \end{array} \tag{5}$$

```
   do iB=1,M,N
     do jn=1,MIN(iB+N-1,M)
       do in=MAX(iB,jn),MIN(iB+N-1,M)
S1:      A(in,jn) = A(jn,in)
       enddo
     enddo
   enddo
```

$$\exists t \; s.t. \quad \begin{array}{c} i_b = tN + 1 \\ 2 - N \le i_b \le M \\ M \ge 1 \wedge N \ge 1 \\ t \ge 0 \end{array} \tag{6}$$

Figure 3: Symbolic blocking example

1.1 Rectangular symbolic linearization

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*

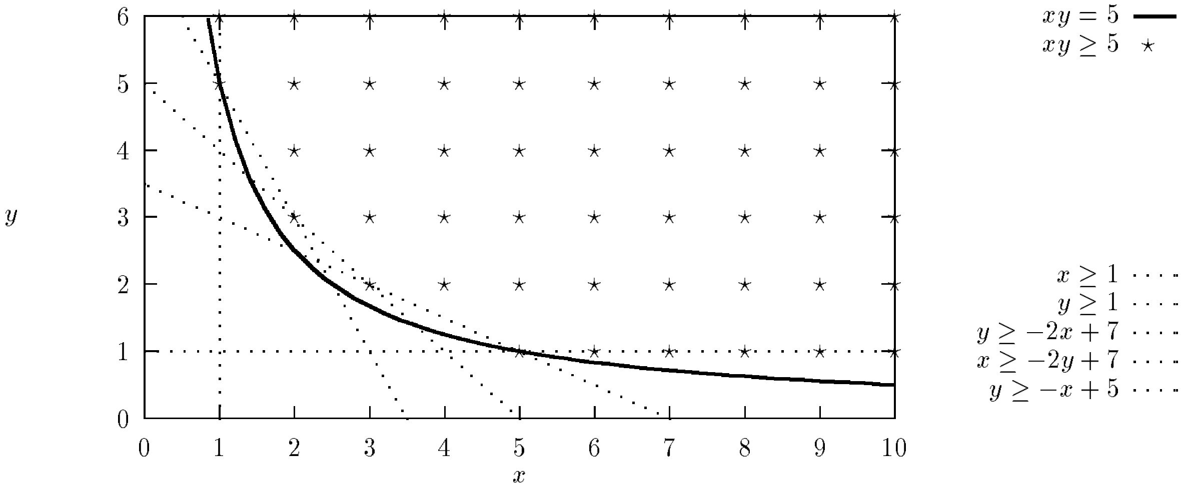

Figure 4: Affinizing inequality $xy \geq 5$

[KP93]. The mapping from the old iteration space to the new iteration space is described by a relation

$$S_1 : [i_o, j_o] \rightarrow [i_b, j_n, i_n] \mid i_b = \lfloor (i_o - 1)/N \rfloor N + 1 \wedge j_n = j_o \wedge i_n = i_o \quad (7)$$

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 `iB`. 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*:

$$\begin{array}{rl} xy \geq c & \text{A hyperbolic inequality} \\ xy = c & \text{A hyperbolic equality} \\ a_x x^2 + a_y y^2 \geq c & \text{An elliptical inequality} \\ a_x x^2 + a_y y^2 = c & \text{An elliptical equality} \end{array} \quad (8)$$

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):

$$(x \geq 1 \wedge 2x+y \geq 7 \wedge x+y \geq 5 \wedge x+2y \geq 7 \wedge y \geq 1) \vee (x \leq -1 \wedge 2x+y \leq -7 \wedge x+y \leq -5 \wedge x+2y \leq -7 \wedge y \leq -1)$$

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 ∧ change)
        change := False
        v := affine variables of p_n
        p_{n+1} := ∃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 ∧ q_n ∧ q_{n-1} ∧ ··· ∧ 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

$$Nj_w + k_w = Nj_r + k_r \ \wedge \ 1 \le j_w N - Nj_r + N \ \wedge \ q \le j_w, j_r \le q + M - 1$$

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

$$v_1 + k_w = v_2 + k_r \ \wedge \ 1 \le v_1 - v_2 + N \ \wedge \ q \le j_w, j_r \le q + M - 1$$
$$v_1 := Nj_w, \quad v_2 := Nj_r$$

Please note, that products Nj_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

$$\exists\ y\ s.t.\ \begin{array}{rcl} y & = & \sum_{i=1}^{n} a_i R_i \\ xy & \geq & c \end{array}$$

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):

$$\exists\ \alpha, \beta\ s.t.\ \begin{array}{rcl} \alpha & = & 2a_x^2 a_y x - 2a_x a_y^2 y + b_x a_y - b_y a_x \\ \beta & = & 2a_x^2 a_y x + 2a_x a_y^2 y + b_x a_y + b_y a_x \\ \alpha\beta & \geq & a_y^2 b_x^2 - a_x^2 b_y^2 - 4a_x^2 a_y^2 c \end{array}$$

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):

$$\exists\ \alpha, \beta\ s.t.\ \begin{array}{lcl} \alpha & = & 2a_x x + b_x \\ \beta & = & 2a_y y + b_y \\ a_y \alpha^2 + a_x \beta^2 & \geq & a_y b_x^2 + a_x b_y^2 - 4a_x a_y c \end{array}$$

5 Representing integer division.

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

$$L(\lfloor E/F \rfloor, ...)$$

to polynomial constraint:

$$\exists\ t, \alpha\ s.t.\ \ L(t, ...) \wedge tF + \alpha = E \wedge 0 \leq \alpha \leq F - 1$$

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):

$$x_b \leq x \leq x_e \wedge y \leq f(x) \tag{9}$$

We require this area to be *convex*, that is, $\forall x\colon x_b \leq x \leq x_e \Rightarrow f''(x) \leq 0$. If it is not so, we can break the segment $[x_b, x_e]$ 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 $[x_b, x_e]$ into 4 intervals (some of them can be empty):

$$\begin{array}{lcrcl} x_b \leq x \leq x_1 & \Rightarrow & 1 \leq & f'(x) & \\ x_1 \leq x \leq x_0 & \Rightarrow & 0 \leq & f'(x) & \leq 1 \\ x_0 \leq x \leq x_{-1} & \Rightarrow & -1 \leq & f'(x) & \leq 0 \\ x_{-1} \leq x \leq x_e & \Rightarrow & & f'(x) & \leq -1 \end{array}$$

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

$$\begin{array}{l} \lceil x_b \rceil \leq x \leq \lfloor x_e \rfloor \wedge y \leq \lfloor f(x_0) \rfloor \wedge \\ \displaystyle\bigwedge_{i=\lceil x_b \rceil}^{\lfloor x_1 \rfloor} \mathsf{line}\,(\langle i, \lfloor f(i) \rfloor \rangle,\ \langle i+1, \lfloor f(i+1) \rfloor \rangle) \leq 0 \wedge \bigwedge_{i=\lceil f(x_1) \rceil}^{\lfloor f(x_0)-1 \rfloor} \mathsf{line}\,(\langle \lceil f_+^{-1}(i) \rceil, i \rangle,\ \langle \lceil f_+^{-1}(i+1) \rceil, i+1 \rangle) \leq 0 \wedge \\ \displaystyle\bigwedge_{i=\lceil f(x_{-1}) \rceil}^{\lfloor (x_0)-1 \rfloor} \mathsf{line}\,(\langle \lfloor f_-^{-1}(i) \rfloor, i \rangle,\ \langle \lfloor f_-^{-1}(i+1) \rfloor, i+1 \rangle) \geq 0 \wedge \bigwedge_{i=\lceil x_{-1} \rceil}^{\lfloor x_e \rfloor} \mathsf{line}\,(\langle i-1, \lfloor f(i-1) \rfloor \rangle,\ \langle i, \lfloor f(i) \rfloor \rangle) \leq 0 \end{array} \tag{10}$$

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 $\mathsf{line}\,(\langle x_1, y_1 \rangle,\ \langle x_2, y_2 \rangle)$ gives back an expression that is zero along the straight line passing through the points $\langle x_1, y_1 \rangle$ and $\langle x_2, y_2 \rangle$, positive to the left of that line (as we move from $\langle x_1, y_1 \rangle$ to $\langle x_2, y_2 \rangle$) and negative to the right of that line:

$$\mathsf{line}\,(\langle x_1, y_1 \rangle,\ \langle x_2, y_2 \rangle) = (x_2 - x_1)(y - y_1) - (y_2 - y_1)(x - x_1)$$

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: $xy \geq c \geq 1 \equiv (x \leq -1 \wedge y \leq c/x) \vee (x \geq 1 \wedge y \geq c/x)$. Applying the affinization theorem to the each area, we get that the inequality $xy \geq c \geq 1$ is equivalent to:

$$x \geq 1 \wedge y \geq 1 \wedge \bigwedge_{i=1}^{\lceil\sqrt{c}-1\rceil} (\mathsf{line}\,(\langle i, \left\lceil \frac{c}{i} \right\rceil\rangle,\ \langle i+1, \left\lceil \frac{c}{i+1} \right\rceil\rangle) \geq 0 \wedge \mathsf{line}\,(\langle \left\lceil \frac{c}{i} \right\rceil, i\rangle,\ \langle \left\lceil \frac{c}{i+1} \right\rceil, i+1\rangle) \leq 0) \quad \vee$$

$$x \leq -1 \wedge y \leq -1 \wedge \bigwedge_{i=\lfloor 1-\sqrt{c}\rfloor}^{-1} (\mathsf{line}\,(\langle i, \left\lfloor \frac{c}{i} \right\rfloor\rangle,\ \langle i-1, \left\lfloor \frac{c}{i-1} \right\rfloor\rangle) \geq 0 \wedge \mathsf{line}\,(\langle \left\lfloor \frac{c}{i} \right\rfloor, i\rangle,\ \langle \left\lfloor \frac{c}{i-1} \right\rfloor, i-1\rangle) \leq 0)$$

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

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:

$$xy \geq c \;\equiv\; \neg(x'y' \geq c') \;\text{ where }\; x' = -x,\; y' = y,\; c' = 1 - c \geq 1$$

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 $a_x x^2 + a_y y^2 = c$ 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 $t = 1$ up to $\lfloor\sqrt{|c|}\rfloor$, if t divides c then we add to the disjunction constraints:

$$\begin{array}{llllll} (x = t & \wedge & y = c/t) & \vee & (x = c/t & \wedge & y = t) & \vee \\ (x = -t & \wedge & y = -c/t) & \vee & (x = -c/t & \wedge & y = -t) & \end{array}$$

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

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):

$$L_x = \min x,\; L_y = \min y,\; U_x = \max x,\; U_y = \max y$$

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 $x \geq 1 \wedge x + y \geq 3 \wedge y \geq 1$.

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:

$$p_1 \equiv \begin{array}{c} 1 - N \leq MNi_w + Nj_w - MNi_r - Nj_r \leq N - 1 \\ 1 - M \leq j_w - j_r \leq M - 1 \end{array}, \quad q_1 \equiv \begin{array}{c} MNi_w + Nj_w + k_w = MNi_r + Nj_r + k_r \\ p \leq i_w, i_r \leq p + L - 1 \\ q \leq j_w, j_r \leq q + M - 1 \\ r \leq k_w, k_r \leq r + N - 1 \end{array}$$

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:

$$\begin{array}{c} 1 \leq N(Mi_w - Mi_r + j_w - j_r + 1) \\ 1 \leq N(Mi_r - Mi_w + j_r - j_w + 1) \\ 1 - M \leq j_w - j_r \leq M - 1 \end{array} \equiv \begin{array}{c} 1 \leq N \\ 1 \leq Mi_w - Mi_r + j_w - j_r + 1 \\ 1 \leq Mi_r - Mi_w + j_r - j_w + 1 \\ 1 - M \leq j_w - j_r \leq M - 1 \end{array} \equiv \begin{array}{c} 1 \leq N \\ Mi_w + j_w = Mi_r + j_r \\ 1 - M \leq j_w - j_r \leq M - 1 \end{array}$$

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

$$p_2 \equiv \ 1 - M \leq Mi_r - Mi_w \leq M - 1 \ \equiv \begin{array}{c} 1 \leq M(i_w - i_r + 1) \\ 1 \leq M(i_r - i_w + 1) \end{array}, \quad q_2 \equiv \begin{array}{c} 1 \leq N \\ Mi_w + j_w = Mi_r + j_r \end{array}$$

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

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

$$p \equiv p_2 \ \wedge \ q_2 \ \wedge \ q_1 \equiv \begin{array}{c} i_w = i_r \\ Mi_w + j_w = Mi_r + j_r \\ MNi_w + Nj_w + k_w = MNi_r + Nj_r + k_r \\ p \leq i_w, i_r \leq p + L - 1 \\ q \leq j_w, j_r \leq q + M - 1 \\ r \leq k_w, k_r \leq r + N - 1 \end{array}$$

Substituting i_r for i_w allows us to derive $j_w = j_r$, which in turn allows us to substitute j_r for j_w deriving $k_w = k_r$:

$$\begin{array}{c} p \leq i_w = i_r \leq p + L - 1 \\ q \leq j_w = j_r \leq q + M - 1 \\ r \leq k_w = k_r \leq r + N - 1 \end{array}$$

7.2 Triangular delinearization

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

$$p \equiv \exists\, t_1, t_2, \alpha, \beta \ s.t. \begin{array}{rcl} 2t_1 + \alpha = i_1^2 + i_1 & \wedge & 0 \leq \alpha \leq 1 \\ 2t_2 + \beta = i_2^2 + i_2 & \wedge & 0 \leq \beta \leq 1 \\ t_1 + j_1 = t_2 + j_2 & \wedge & 0 \leq i_1, i_2 \leq N - 1 \\ 0 \leq j_1 \leq i_1 & \wedge & 0 \leq j_2 \leq i_2 \end{array} \equiv \begin{array}{c} 0 \leq j_2 \leq i_2 < N \\ 0 \leq j_1 \leq i_1 < N \\ i_2 + 2j_2 + i_2^2 \leq 1 + i_1 + 2j_1 + i_1^2 \\ i_1 + 2j_1 + i_1^2 \leq 1 + i_2 + 2j_2 + i_2^2 \end{array}$$

We now compute p_1, the projection of p onto variables involved in products, and q_1, everything else needed so that $p = p_1 \wedge q_1$:

$$p_1 \equiv \begin{array}{c} 0 \le i_1 \\ 0 \le i_2 \\ i_2 + i_2^2 \le 1 + 3i_1 + i_1^2 \\ i_1 + i_1^2 \le 1 + 3i_2 + i_2^2 \end{array} \qquad q_1 \equiv \begin{array}{c} 0 \le j_2 \le i_2 < N \\ 0 \le j_1 \le i_1 < N \\ i_2 + 2j_2 + i_2^2 \le 1 + i_1 + 2j_1 + i_1^2 \\ i_1 + 2j_1 + i_1^2 \le 1 + i_2 + 2j_2 + i_2^2 \end{array}$$

We factor the polynomial constraints in p_1:

$$\begin{array}{rcl} i_1^2 + 3i_1 - i_2^2 - i_2 \ge 0 & \equiv & (i_1 - i_2 + 1)(i_1 + i_2 + 2) \ge 2 \\ i_2^2 + 3i_2 - i_1^2 - i_1 \ge 0 & \equiv & (i_2 - i_1 + 1)(i_2 + i_1 + 2) \ge 2 \end{array}$$

and affinize the factored forms:

$$\begin{array}{rcl} (i_1 - i_2 + 1)(i_1 + i_2 + 2) \ge 2 & \equiv & i_1 - i_2 + 1 \ge 1 \wedge i_1 + i_2 + 2 \ge 1 \wedge (i_1 - i_2 + 1) + (i_1 + i_2 + 2) \ge 3 \\ (i_2 - i_1 + 1)(i_2 + i_1 + 2) \ge 2 & \equiv & i_2 - i_1 + 1 \ge 1 \wedge i_2 + i_1 + 2 \ge 1 \wedge (i_2 - i_1 + 1) + (i_2 + i_1 + 2) \ge 3 \end{array}$$

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

$$p_1 \equiv 0 \le i_1 = i_2$$

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

$$p \equiv p_1 \ \wedge \ q_1 \equiv \begin{array}{c} i_1 = i_2 \\ 0 \le j_2 \le i_2 < N \\ 0 \le j_1 \le i_1 < N \\ i_2 + 2j_2 + i_2^2 \le 1 + i_1 + 2j_1 + i_1^2 \\ i_1 + 2j_1 + i_1^2 \le 1 + i_2 + 2j_2 + i_2^2 \end{array}$$

By substituting i_1 for i_2, we get:

$$p \equiv \begin{array}{c} i_2 := i_1 \\ 0 \le j_2 \le i_1 < N \\ 0 \le j_1 \le i_1 < N \\ i_1 + 2j_2 + i_1^2 \le 1 + i_1 + 2j_1 + i_1^2 \\ i_1 + 2j_1 + i_1^2 \le 1 + i_1 + 2j_2 + i_1^2 \end{array} \equiv \begin{array}{c} i_2 := i_1 \\ 0 \le j_2 \le i_1 < N \\ 0 \le j_1 \le i_1 < N \\ 2j_2 \le 1 + 2j_1 \\ 2j_1 \le 1 + 2j_2 \end{array} \equiv \begin{array}{c} i_2 := i_1 \\ 0 \le j_2 \le i_1 < N \\ 0 \le j_1 \le i_1 < N \\ j_2 \le j_1 \\ j_1 \le j_2 \end{array} \equiv \begin{array}{c} 0 \le j_1 \le i_1 < N \\ i_2 = i_1 \\ j_2 = j_1 \end{array}$$

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

$$IS = 1 \le j_o \le i_o \le M$$

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

$$i_b = \lfloor (i_o - 1)/N \rfloor N + 1$$

used in the schedule to polynomial constraint

$$\exists \ t \ s.t. \ i_b = tN + 1 \ \wedge \ tN + \alpha = i_o - 1 \ \wedge \ 0 \le \alpha \le N - 1$$

So constraints on new variables i_b, j_n, i_n are:

$$IS' = \exists \ i_o, j_o, t, \alpha \ s.t. \ \begin{array}{c} 1 \le j_o \le i_o \le M \\ i_b = tN + 1 \\ tN + \alpha = i_o - 1 \\ 0 \le \alpha \le N - 1 \\ j_n = j_o \ \wedge \ i_n = i_o \end{array}$$

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:

$$p_1 \equiv N + tN \geq 1 \ \wedge \ N \geq 1 \quad \equiv \quad N(t+1) \geq 1 \ \wedge \ N \geq 1 \quad \equiv \quad t \geq 0 \ \wedge \ N \geq 1$$

  Adding p_1 to the original problem and simplifying produces:

$$IS' = \exists\ t\ s.t. \quad \begin{array}{c} 1 \leq j_n \leq i_n \leq M \\ i_b = tN + 1 \\ tN < i_n \leq N + tN \\ t \geq 0 \end{array}$$

  Projecting IS' onto i_b and symbolic constants produces:

$$p_{i_b} \equiv \exists t\ s.t. \quad \begin{array}{c} i_b = tN + 1 \\ 2 - N \leq i_b \leq M \\ M \geq 1 \ \wedge \ N \geq 1 \\ t \geq 0 \end{array} \tag{11}$$

  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 $expr \leq i_b$ 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:

$$\exists i_b\ s.t. \quad \begin{array}{c} i_b = tN + 1 \\ 2 - N \leq i_b \leq M \\ M \geq 1 \ \wedge \ N \geq 1 \\ t \geq 0 \end{array} \quad \equiv \quad \begin{array}{c} 2 - N \leq N + 1 \leq M \\ M \geq 1 \ \wedge \ N \geq 1 \\ t \geq 0 \end{array} \quad \equiv \quad \begin{array}{c} 0 \leq t \\ tN \leq M - 1 \\ M \geq 1 \ \wedge \ N \geq 1 \end{array} \tag{12}$$

  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 $t \star 0 + 1 = 1$.

  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 $\pi_{M,N,i_b,j_n,t,tN}(IS')$ given (11). Simplifying we get:

$$1 \leq j_n \leq M \ \wedge \ j_n \leq N + i_b - 1 \tag{13}$$

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

$$j_n \leq i_n \leq M \ \wedge \ i_b \leq i_n \leq N + i_b - 1 \tag{14}$$

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:

$$\begin{array}{c} Ne_1 + e_2 = 0 \\ 1 - N \le e_2 \le N - 1 \end{array} \quad \text{to} \quad \begin{array}{c} e_1 = 0 \\ e_2 = 0 \end{array}$$

where e_1, e_2 are expressions, and N is a variable. Using our algorithm, we substitute $e_2 = -Ne_1$ into the inequality $1 - N \le e_2 \le N - 1$. Factoring and simplifying produces: $N(1 - e_1) \ge 1 \ \wedge \ N(1 + e_1) \ge 1$. Affinizing both inequalities we get $e_1 \ge 0 \wedge e_1 \le 0$ and therefore $e_1 = 0$. Substituting this equality to the original problem, we finally prove that $e_1 = e_2 = 0$.

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 iteration space.

Parafrase-2. In [HP91] the authors propose to use a symbolic version of Banerjee's inequalities for dependence 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 affinizing 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 `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

[AK87] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[Buc85] B. Buchberger. Grobner bases: an algorithmic method in polynomial ideal theory. In N. K. Bose, editor, *Multidimensional Systems Theory*. D. Reidel Publishing Co., 1985.

[DST88] J. H. Davenport, Y. Siret, and E. Tournier. *Computer Algebra, Systems and Algorithms for Algebraic Computation*. Academic Press, 1988.

[HP91] M. Haghighat and C. Polychronopoulos. Symbolic dependence analysis for high-performance parallelizing compilers. In *Advances In Languages And Compilers for Parallel Processing*, August 1991.

[HP93] M. Haghighat and C. Polychronopoulos. Symbolic analysis: A basis for parallelization, optimization and scheduling of programs. In *Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[KL92] Deepak Kapur and Yagiti Lakshman. Elimination methods: an introduction. In Bruce Donald, Deepak Kapur, and Joseph Mundy, editors, *Symbolic and Numerical Computation for Artificial Intelligence*. Academic Press, 1992.

[KP93] Wayne Kelly and William Pugh. A framework for unifying reordering transformations. Technical Report CS-TR-3193, Dept. of Computer Science, University of Maryland, College Park, April 1993.

[Mas92] Vadim Maslov. Delinearization: an efficient way to break multiloop dependence equations. In *ACM SIGPLAN '92 Conf. on Programming Language Design and Implementation*, San Francisco, California, June 1992.

[Pug92] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, August 1992.

[PW92] William Pugh and David Wonnacott. Going beyond integer programming with the Omega test to eliminate false data dependences. Technical Report CS-TR-3191, Dept. of Computer Science, University of Maryland, College Park, December 1992. An earlier version of this paper appeared at the SIGPLAN PLDI'92 conference.

[PW93] William Pugh and David Wonnacott. An evaluation of exact methods for analysis of value-based array data dependences. In *Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.