UMIACS-TR-94-27 March, 1994
CS-TR-3234

Counting Solutions to Presburger Formulas: How and Why

William Pugh
pugh@cs.umd.edu
http://www.cs.umd.edu/faculty/pugh.html
Institute for Advanced Computer Studies
Dept. of Computer Science
Univ. of Maryland, College Park, MD 20742

We describe methods that are able to count the number of integer solutions to selected free variables of a Presburger formula, or sum a polynomial over all integer solutions of selected free variables of a Presburger formula. This answer is given symbolically, in terms of symbolic constants (the remaining free variables in the Presburger formula).

For example, we can create a Presburger formula who's solutions correspond to the iterations of a loop. By counting these, we obtain an estimate of the execution time of the loop.

In more complicated applications, we can create Presburger formulas who's solutions correspond to the distinct memory locations or cache lines touched by a loop, the flops executed by a loop, or the array elements that need to be communicated at a particular point in a distributed computation. By counting the number of solutions, we can evaluate the computation/memory balance of a computation, determine if a loop is load balanced and evaluate message traffic and allocate message buffers.

This work is supported by an NSF PYI grant CCR-9157384 and by a Packard Fellowship.

1 Introduction

In this paper, we describe methods that are able to count the number of integer solutions to selected free variables of a Presburger formula, or sum a polynomial over all integer solutions of selected free variables of a Presburger formula. This answer is given symbolically, in terms of symbolic constants (the remaining free variables in the Presburger formula). This answer can be given symbolically, in terms of symbolic constants (other free variables in the Presburger formula). Presburger formulas are those formulas that can be built up out of linear constraints over integer variables, the usual logical connectives $\wedge$, $\vee$, $\neg$, and existential and universal quantifiers.

The following table gives some simple examples of traditional symbolic summations:

Sum	Answer	
$\sum_{i=1}^{10} 1$	10	
$\sum_{i=1}^{n} 1$	n	(if $n \geq 1$)
$\sum_{i=1}^{n} \sum_{j=1}^{n} 1$	n^2	(if $n \geq 1$)
$\sum_{i=1}^{n} \sum_{j=i+1}^{n} 1$	$\frac{n(n-1)}{2}$	(if $n \geq 2$)

A number of symbolic math packages (such as Mathematica and Maple) are able to compute symbolic sums. However, the methods they use assume that a summation is never empty (i.e., the lower bound is never greater than the upper bound). The answer they give is incorrect if this assumption is violated. For example, Mathematica reports that

$$\sum_{i=1}^{n} \sum_{j=i}^{m} 1 = \sum_{i=1}^{n} (m - i + 1) = \frac{n(2m - n + 1)}{2}$$

In fact, this answer is valid only if $1 \leq n \leq m$. If $1 \leq m < n$, the answer is $m(m+1)/2$.

The notation used above strongly suggests that the sum over j must be performed before the sum over i. In our work, we don't assume any predefined order in which the variables must be eliminated. We also allow arbitrary constraints, not just a upper and lower bound on each variable. We therefore use a more general notation:

$$(\Sigma v_1, \ldots, v_m \ : \ P \ : \ x)$$

is the sum, for all values of $v_1, \ldots, v_m$ that satisfy P (a Presburger formula with free variables), of x. As a special case, $(\Sigma v_1, \ldots, v_m \ : \ P \ : \ 1)$ is the number of solutions of $v_1, \ldots, v_m$ to P. Any variables appearing free in P or x and not in V are assumed to be symbolic constants. The answer returned will be in terms of the symbolic constants.

Since answers may need to be guarded (i.e., n^2 if $n \geq 1$), we utilize the nullary form of a summation to indicate a guarded sum. If V is empty (i.e., $(\Sigma \ : \ P \ : \ x)$) the result is a conditional value: if P is true, the value of this expression is x, otherwise 0. As an example, our previous summations would be reported as:

$$(\Sigma i \ : \ 1 \leq i \leq 10 \ : \ 1) = 10$$

$$(\Sigma i \ : \ 1 \leq i \leq n \ : \ 1) = (\Sigma \ : \ 1 \leq n \ : \ n)$$

$$(\Sigma i, j \ : \ 1 \leq i, j \leq n \ : \ 1) = (\Sigma \ : \ 1 \leq n \ : \ n^2)$$

$$(\Sigma i, j \ : \ 1 \leq i < j \leq n \ : \ 1) = (\Sigma \ : \ 2 \leq n \ : \ \frac{n(n-1)}{2})$$

1.1 Applications

This capability has a number of applications in analysis and transformations of scientific programs. Within programs with affine loop bounds, guards and subscripts, we can define formulas who's solutions correspond to:

- the flops executed by a loop
- the memory locations touched by a loop
- the cache lines touched by a loop
- the array elements that need to be transmitted from one processor to another during the execution of a loop (in a distributed memory multicomputer).

By counting the number of solutions to these formulas, we can

- estimate the execution time of a code segment,
- compare the memory bandwidth requirements vs. the flop counts of a code segment,
- determine which loops will flush the cache, allowing us to calculate the cache miss rate [FST91],
- determine whether a parallel loop is load balanced (i.e., does each iteration perform the same number of flops) [TF92],
- given an unbalanced loop, assign different number of iterations to each processor so that each processor gets the same total number of flops (Balanced chunk-scheduling, as described in [HP93a]),
- quantify message traffic, and
- allocate space for message buffers.

In this paper, we

- Review the Omega test and how it can be used to simplify Presburger formulas (Section 2).
- Describe how to represent some nonlinear constraints, such as $x = \lfloor y/3 \rfloor$, in Presburger formulas (Section 3).

- Describe techniques for computing sums, starting with the most simple forms of summations and progressing to more general sums (Section 4).
- Describe techniques for producing simplified constraints in *disjoint* disjunctive normal form. Previously, we have produced simplified constraints in (overlapping) disjunctive normal form. The need to do this is explained in Section 4.5.1 and techniques to do it are described in Section 5.
- Show the application of our techniques to a number of examples and compare our techniques with relation work [FST91, TF92, HP93a, Taw94] (Section 6).

2 The Omega test

The Omega test [Pug92] was originally developed to check if a set of linear constraints had an integer solution, and was initially used in array data dependence testing. Since then, its capabilities and uses have grown substantially. In this section, we describe the various capabilities of the Omega test.

2.1 Eliminating an existentially quantified variable

The basic operation of the Omega test is the elimination of an existentially quantified variable, also refered to as shadow-casting or projection. For example, given a set of constraints P over x, y and z that define, for example, a dodecahedron, the Omega test can compute the constraints on x and y that define the shadow of the dodecahedron. Mathematically, these constraints are equivalent to $\exists z$ s.t. P. But the Omega test is able to remove the existentially quantified variables, and report the answer just in terms of the free variables (x and y).

Over rational variables, projection of a convex region always gives a convex result. Unfortunately, the same does not apply for integer variables. For example, $\exists y$ s.t. $1 \leq y \leq 4 \wedge x = 2y$ has $x = 2$, $x = 4$, $x = 6$ and $x = 8$ as solutions. Sometimes, the result is even more complicated. For example, the solutions for x in:

$$(\exists i, j \ : \ 1 \leq i \leq 8 \wedge 1 \leq j \leq 5 \wedge x = 6i + 9j - 7)$$

are all numbers between 8 and 86 (inclusive) that have remainder 2 when divided by 3, except for 11 and 83. In general, the Omega test produces an answer in disjunctive normal form: the union of a list of clauses. A clause may need to describe a nonconvex region. There are two methods of describing these regions:

Stride format The Omega test can produce clauses that consist of affine constraints over the free variables and stride constraints. A stride constraint $c|e$ is interpreted as "c evenly divides e". In this form, the above solution could be represented as:

$$x = 8 \ \vee \ 14 \leq x \leq 80 \wedge 3|(x+1) \ \vee \ x = 86$$

Projected format Alternatively, the Omega test can produce clauses that consist of a set of linear constraints over a set of auxiliary variables and an affine 1-1 mapping from those variables to the free variables. Using this format, the above solution could be represented as

$$x = 8 \vee (\exists \alpha \ : \ 5 \leq \alpha \leq 27 \wedge x = 3\alpha - 1) \vee x = 86$$

These two representations are equivalent and there are simple and efficient methods for converting between the two of them. While the first representation is more intuitive, the second representation works better for the purposes of this paper.

Disjoint disjunctive normal form Normally, the Omega test does not produce disjoint clauses: there may be assignments to the free variables that satisfy multiple clauses (i.e., the clauses may overlap). For purposes of this paper, it is preferable to have disjoint clauses. This allows us to compute the summation by simply adding together the results of summing the individual clauses. If a set of clauses are disjoint, we refer to this as disjoint disjunctive normal form. If two clauses are *guaranteed* to be disjoint, we denote their conjunction as $P + Q$ (as opposed to $P \vee Q$).

In our previous work, we did not need disjoint clauses. Some straightforward but naive methods would be capable of converting an arbitrary disjunction normal form formula into disjoint disjunctive normal form. However, the cost of doing so would be quite high in many cases.

In Section 5, we discuss some methods that allow us to directly generate disjoint disjunctive normal form (without first generating overlapping disjunctive normal form) and more sophisticated methods for converting an arbitrary disjunctive normal form formula into a disjoint disjunctive normal form. Disjoint disjunctive normal form may generate more clauses and require more time to generate than disjunctive normal form.

2.2 Verifying the Existence of Solutions

The Omega test also provides direct support for checking if integer solutions exist to a set of linear constraints. It does this by treating all the variables as existentially quantified and eliminating variables until it produces a problem containing a single variable; such problems are easy to check for integer solutions. The Omega test incorporates several extensions over a naive application of variable elimination.

2.3 Removing Redundant Constraints

In the normal operation of the Omega test, we eliminate any constraint that is made redundant by any other single constraint (e.g., $x + y \le 10$ is made redundant by $x + y \le 5$). Upon request, we can use more aggressive techniques to eliminate redundant constraints. We use fast but incomplete tests that can flag a constraint as definitely redundant or definitely not redundant, and a backup complete test. This capability is used when verifying implications and simplifying formulas involving negation.

We also use these techniques to define a "gist" operator, which is defined such that (gist P given Q) is what is "interesting" about P, given that we already know Q. In other words, we guarantee that ((gist P given Q) $\wedge$ Q) $\equiv P \wedge Q$ and try to make the result of the gist operator as simple as possible. More formally, gist P given Q returns a subset of the constraint of P such that none of the constraints returned are implied by the constraints of Q and the other constraints in the result.

2.4 Verifying Implications

By using our ability to eliminate redundant constraints, we can verify formulas of the form $P \Rightarrow Q$, by checking to see if the constraints of P are redundant, given that the constraints of Q are true. (i.e., (gist P given Q) $\equiv$ True). We can combine this capability with our ability to eliminate existentially quantified variables to verify more complicated formulas such as ($\exists y$ s.t. P) $\Rightarrow$ ($\exists z$ s.t. Q).

2.5 Simplifying Formulas Involving Negation

There are two problems involved in simplifying formulas containing negated conjuncts, such as

$$-10 \le i + j, i - j \le 10 \ \wedge\ \neg(2 \le i, j \le 12 \wedge 2|i + j)$$

Naively converting such formulas to disjunctive normal form generally leads to an explosive growth in the size of the formula. In the worst-case, this cannot be prevented. But we [PW93a] have described methods that are effective in dealing with these problems for the cases we encounter. Secondly, previous techniques for negating non-convex constraints, based on quasilinear constraints [AI91], were discovered to be incomplete in certain pathological cases [PW93a]. We [PW93a] describe a method that is exact and complete for all cases.

2.6 Simplifying Arbitrary Presburger Formulas

Utilizing the capabilities described above, we can simplify and/or verify arbitrary Presburger formulas. In general, this may be prohibitively expensive. There is a nondeterministic lower bound of $2^{2^{o(n)}}$ and a deterministic upper bound of $2^{2^{2^{O(n)}}}$ on the time required to verify a Presburger formula. However, we have found that we are able to efficiently analyze many Presburger formulas that arise in practice.

For example, our current implementation requires 12 milliseconds on a Sun Sparc IPX to simplify

$$\begin{array}{c}
1 \le \mathrm{i} \le 2\mathtt{n} \ \wedge\ 1 \le \mathrm{i}'' \le 2\mathtt{n} \wedge \mathrm{i} = \mathrm{i}'' \\
\wedge\ \neg(\ \exists \mathrm{i}', \mathrm{j}' \text{ s.t. } 1 \le \mathrm{i}' \le 2\mathtt{n} \wedge 1 \le \mathrm{j}' \le \mathtt{n} - 1 \\
\wedge \mathrm{i} \le \mathrm{i}' \wedge \mathrm{i}' = \mathrm{i}'' \wedge 2\mathrm{j}' = \mathrm{i}''\ ) \\
\wedge\ \neg(\ \exists \mathrm{i}', \mathrm{j}' \text{ s.t. } 1 \le \mathrm{i}' \le 2\mathtt{n} \wedge 1 \le \mathrm{j}' \le \mathtt{n} - 1 \\
\wedge \mathrm{i} \le \mathrm{i}' \wedge \mathrm{i}' = \mathrm{i}'' \wedge 2\mathrm{j}' + 1 = \mathrm{i}''\ )
\end{array}$$

to

$$(1 = \mathrm{i} = \mathrm{i}'' \le \mathtt{n}) \vee (1 \le \mathrm{i} = \mathrm{i}'' = 2\mathtt{n})$$

3 Nonlinear constraints

Generally, Presburger formulas are thought to allow only linear constraints. It turns out that there are a number of nonlinear constraints that can be supported while remaining in the class of Presburger formulas.

3.1 Floors, ceilings and mods

If a term $\lfloor x/c \rfloor$ appears in a constraint C, we replace C with

$$\exists \alpha \text{ s.t. } c\alpha \le x < c(\alpha + 1) \wedge C'$$

where C' is C with $\lfloor x/c \rfloor$ replaced with α.

If a term $\lceil x/c \rceil$ appears in a constraint C, we replace C with

$$\exists \beta \text{ s.t. } c(\beta - 1) < x \le c\beta \wedge C'$$

where C' is C with $\lfloor x/c \rfloor$ replaced with β.

If a term $x \bmod c$ appears in a constraint C, we replace C with

$$\exists \gamma \text{ s.t. } c\gamma \le x < c(\gamma + 1) \wedge C'$$

where C' is C with $x \bmod c$ replaced with $x - \gamma$.

3.2 Stride constraints

A stride constraint $c|e$ requires that e be evenly divisible by c. This is equivalent to $\exists \alpha$ s.t. $e = c\alpha$. A negated stride constraint $\neg(c|e)$ requires that e not be evenly divisible by c. This is equivalent to $\exists \alpha$ s.t. $c\alpha < e < c(\alpha + 1)$.

3.3 Applications

Among other applications, nonlinear constraints of this form show up in analyzing and compiling HPF code with distributed arrays.

Assume a one dimensional template `T(0:1024)` has been distributed in block-cyclic fashion to 8 processors, using blocks of 4. This means elements `T(0:3)` are mapped to processor 0, `T(4:7)` to processor 1, `T(28:31)` to processor 7, and `T(32:35)` to processor 0 again. This can be described as a mapping from a template index t to a processor number p and an index $[c, l]$ of a two-dimensional array of local data. This mapping can be described as:

$$l = t \bmod 4 \qquad p = \left\lfloor \frac{t}{4} \right\rfloor \bmod 8 \qquad c = \left\lfloor \frac{t}{32} \right\rfloor$$

which is equivalent to

$$t = l + 4p + 32c \wedge 0 \le l \le 3 \wedge 0 \le p \le 7$$

4 Computing sums

In this section, we describe our methods for computing sums. In computing a sum, there are two places where we are given the option of computing the sum exactly, or computing upper and lower bounds. Performing the calculation exactly will likely be more expensive. Also, if the answer is symbolic, an exact answer may be more complicated and harder to utilize. It may often be preferable to compute both an upper and lower bound on the sum. Only if these values are far apart may it be worthwhile to compute the exact answer.

4.1 Simple sums

There are fairly standard formulas for sums of powers of integers. These formulas are described in the CRC Standard Mathematical Tables [Bey81] and are reviewed in [TF92, Taw94]. For example,

$$(\Sigma i \ : \ 1 \le i \le n \ : \ i^2) = (\Sigma \ : \ 1 \le n \ : \ \frac{n(n+1)(2n+1)}{6})$$

Within our implementation, we expect it will be sufficient to hard code the formulas for p up to 10. In each of these sums, the guard produced is $1 \le n$.

4.2 Basic sums

In this section, we concern ourselves with the more general problem of computing

$$(\Sigma i \ : \ \lceil A/a \rceil \le i \le \lfloor B/b \rfloor \ : \ i^p)$$

where a, b and p are known nonnegative integer constants and A and B are integer expressions (equivalently, variables).

There are three issues we need to address: lower bounds other than 1, negative upper or lower bounds, and the cases when a or b is not 1. If p is equal to zero, the sum is simply:

$$(\Sigma \ : \ L \le U \ : \ U - L + 1)$$

If p is greater than zero, we handle the first two issues by breaking down the sum into four pieces:

$$(\Sigma i \ : \ L \le i \le U \ : \ i^p)$$

$$\begin{aligned}
&\equiv (\Sigma i \ : \ 1 \le i \le U \wedge L \le U \ : \ i^p) \\
&\quad -(\Sigma i \ : \ 1 \le i \le L - 1 < U \ : \ i^p) \\
&\quad +(\Sigma i \ : \ L \le i \le -1 \wedge L \le U \ : \ i^p) \\
&\quad -(\Sigma i \ : \ L < U + 1 \le i \le -1 \ : \ i^p)
\end{aligned}$$

$$\begin{aligned}
&\equiv (\Sigma i \ : \ 1 \le i \le U \wedge L \le U \ : \ i^p) \\
&\quad -(\Sigma i \ : \ 1 \le i \le L - 1 < U \ : \ i^p) \\
&\quad +(\Sigma i \ : \ 1 \le i \le -L \wedge L \le U \ : \ (-i)^p) \\
&\quad -(\Sigma i \ : \ 1 \le i \le -U - 1 < -L \ : \ (-i)^p)
\end{aligned}$$

$$\begin{aligned}
&\equiv (\Sigma i \ : \ 1 \le i \le U \wedge L \le U \ : \ i^p) \\
&\quad -(\Sigma i \ : \ 1 \le i \le L - 1 < U \ : \ i^p) \\
&\quad +(-1)^p(\Sigma i \ : \ 1 \le i \le L \le U \ : \ i^p) \\
&\quad -(-1)^p(\Sigma i \ : \ 1 \le i \le -U - 1 < -L \ : \ i^p)
\end{aligned}$$

4.2.1 Handling rational bounds

To compute a summation involving a floor or ceiling, such as

$$(\Sigma i \ : \ 1 \le i \le \lfloor \frac{U}{u} \rfloor \ : \ i^p)$$

we have three options: compute symbolic answers, compute approximate answers, or splinter the problem (i.e., break it up into subcases) so as to produce exact bounds. For each of these techniques, we will consider the example problem:

$$(\Sigma i \ : \ 1 \le i \le \lfloor \frac{n}{3} \rfloor \ : \ i)$$

Compute symbolic answers We can simply introduce a variable for $\lfloor U/u \rfloor$, and produce a result in terms of this variable. In this case, the answer would be

$$\frac{\lfloor n/3 \rfloor (\lfloor n/3 \rfloor + 1)}{2}$$

The problem with this answer is that we cannot symbolically sum it over n. Therefore, we can only produce symbolic answers if the upper bound is a function only of symbolic constants.

An eventual answer that involved terms such as $\lfloor n/3 \rfloor$, $\lfloor (n-1)/3 \rfloor$ and n may be hard to analyze. It may be better to substitute $(U - U')/u$ for $\lfloor U/u \rfloor$, where U' is a new variable defined to be $U \bmod u$. Since $0 \le U' < u$, we can find the most significant terms of the answer by just looking for the highest powers of n. For this example, this gives:

$$\frac{(n - n \bmod 3)(n + 3 - n \bmod 3)}{18}$$

$$\equiv \frac{n^2 + 3n}{18} - (n \bmod 3)\frac{2n + 3 - (n \bmod 3)}{18}$$

Computing approximate answers We can calculate three kinds of approximate answers: upper bounds, lower bounds or best-guess answers. For any positive integers U and u:

$$\frac{U - (u-1)}{u} \leq \lfloor \frac{U}{u} \rfloor \leq \frac{U}{u} \leq \lceil \frac{U}{u} \rceil \leq \frac{U + (u-1)}{u}$$

So for any function $f(n)$ that is non-decreasing for non-negative n:

$$f(\frac{U - (u-1)}{u}) \leq f(\lfloor \frac{U}{u} \rfloor) \leq f(\frac{U}{u})$$

$$\leq f(\lceil \frac{U}{u} \rceil) \leq f(\frac{U + (u-1)}{u})$$

Since the formula for

$$(\Sigma i \ : \ 1 \leq i \leq n \ : \ i^p)$$

is a non-decreasing function of n, we can use this technique to compute upper and lower bounds on a rational sum.

We can also choose to approximate $f(n)$ as $f(n')$, where n' is the approximate value of n. In our example summation, the lower bound, upper bound and approximation of $\lfloor n/3 \rfloor$ are $(n-2)/3$, $n/3$ and $((n-2)/3 + n/3)/2$ respectively. This gives results of of:

$$\text{lower bound: } (\Sigma \ : \ 3 \leq n \ : \ \frac{(n-2)(n+1)}{18})$$

$$\text{upper bound: } (\Sigma \ : \ 3 \leq n \ : \ \frac{n(n+3)}{18})$$

$$\text{approximation: } (\Sigma \ : \ 3 \leq n \ : \ \frac{(n-1)(n+2)}{18})$$

We could also approximate the answer as the average of the upper and lower bound. Note that this does not give the same answer as the other method of approximation.

Splintering Our other choice is to splinter the problem. Consider the cases $u|U$, $u|(U+1)$, $\ldots$, $u|(U+u-1)$. These cases are all obviously disjoint, so we generate a separate sum for each. Within case, the upper bound on i becomes integral.

$$(\Sigma i \ : \ 3 \mid n \wedge 1 \leq i \leq \frac{n}{3} \ : \ i)$$

$$+(\Sigma i \ : \ 3 \mid n-1 \wedge 1 \leq i \leq \frac{n-1}{3} \ : \ i)$$

$$+(\Sigma i \ : \ 3 \mid n-2 \wedge 1 \leq i \leq \frac{n-2}{3} \ : \ i)$$

$$\equiv (\Sigma \ : \ 3 \mid n \wedge 3 \leq n \ : \ \frac{n(n+3)}{18})$$

$$+(\Sigma \ : \ 3 \mid n-1 \wedge 4 \leq n \ : \ \frac{(n-1)(n+2)}{18})$$

$$+(\Sigma \ : \ 3 \mid n-2 \wedge 5 \leq n \ : \ \frac{(n-2)(n+1)}{18})$$

4.2.2 Producing a guard for rational sums

If both L and U involve floor and ceilings, we may not be able to produce a simple and exact guard to ensure $(\exists i \ : \ L \leq i \leq U)$. If we use the splintering technique described above, our problem is resolved: within each splinter, we will be able to produce a simple and exact guard.

Otherwise, we can choose to splinter the guard or have the Omega test produce a simple but approximate guard. The guard could be an upper or lower bound, depending on whether we are computing an upper and lower bound on the result.

4.3 Polynomial Sums

If the value x we are summing i over is not of the form i^p, we rewrite x as a polynomial of i:

$$(\Sigma i \ : \ L \leq i \leq U \ : \ a_0 + a_1 i + a_2 i^2 + \cdots)$$

$$\equiv a_0(\Sigma i \ : \ L \leq i \leq U \ : \ 1) + a_1(\Sigma i \ : \ L \leq i \leq U \ : \ i)$$
$$+a_2(\Sigma i \ : \ L \leq i \leq U \ : \ i^2) + \cdots$$

4.4 Convex Sums

We now consider a more general form of summation:

$$(\Sigma V \ : \ P \ : \ x)$$

which denotes the sum, for all values of the variables V that satisfy P, of x. For the moment, we require P to be a conjunction of linear inequalities over variables in V and variables representing symbolic constants (i.e., free variables).

Our algorithm for dealing with a convex sum

$$(\Sigma V \ : \ P \ : \ x)$$

is as follows:

1. Eliminate redundant constraints from P.

2. Pick a variable $v \in V$ to consider. In picking a variable, we try to pick a variable who's bounds can be expressed without floors or ceilings, and that has as few upper and lower bounds as possible.

3. If v has multiple upper bounds $U_1, U_2, \ldots, U_p$ in P, we replace the summation with the sum of p new summations, where in the i^{th} summation, the upper bounds on v in P are replaced with:

$$v \leq U_i \leq U_{i+1}, U_{i+2}, \ldots, U_p$$
$$\wedge\ U_i < U_1, U_2, \ldots, U_{i-1}$$

   In each resulting summation, we simplify the resulting conditions (the new, more restrictive constraints might allow us to substantially simplify the conditions), remove redundant constraints and reconsider which variable to eliminate (it might not be v).

4. The case where v has a single upper bound but multiple lower bounds is handled similarly.

5. If v has a single upper bound U and a single lower bound L, compute the sum as

$$(\Sigma V - \{v\}\ :\ P'\ :\ (\Sigma v\ :\ L \leq v \leq U\ :\ x))$$

   Once we have simplified the term $(\Sigma v\ :\ L \leq v \leq U\ :\ x)$, we can simplify the result using the rewrite rules:

$$(\Sigma V\ :\ P\ :\ x+y) \equiv (\Sigma V\ :\ P\ :\ x) + (\Sigma V\ :\ P\ :\ y)$$
$$(\Sigma V\ :\ P\ :\ (\Sigma\ :\ P'\ :\ x)) \equiv (\Sigma V\ :\ P \wedge P'\ :\ x)$$

   If variables remain to be summed over, we will need to repeat this process.

4.5 General Sums

We now consider summations where the conditions P are an arbitrary Presburger formula.

We can use the Omega test, as described in Section 2, to simplify an arbitrary Presburger formula into the union of several clauses. In many cases, the clauses can be described by the conjunction of a set of linear inequalities. In general, a clause may need to be represented as a conjunction of linear inequalities from a set of auxiliary, existentially quantified variables and an affine, 1-1 mapping from the auxiliary variables to variables in V and symbolic constants.

There are three issues we need to deal with:

- Dealing with overlaps between clauses
- Dealing with clauses that are represented by a projection
- Allowing the Omega test to perform approximate simplification

4.5.1 Overlapping clauses

The problem is that some solutions might be solutions to more than one clause. If we sum over the clauses independently, we will count some solutions more than once. One way to handle this, as described in [FST91], is to subtract the count of elements counted twice:

$$(\Sigma V\ :\ P \vee Q\ :\ x)$$
$$\equiv (\Sigma V\ :\ P\ :\ x) + (\Sigma V\ :\ Q\ :\ x) - (\Sigma V\ :\ P \wedge Q\ :\ x)$$

The problem with this is that it quickly gets out of control if there are more than a few clauses (7 summations are needed for 3 clauses). An alternative is to put the formula in disjoint disjunctive normal form, in which the clauses are mutually exclusive. We provide two techniques that let us avoid generating overlapping clauses in disjunctive normal form, and a method that converts an arbitrary formula in disjunctive normal form into disjoint disjunctive normal form. These techniques are described in Section 5.

4.5.2 Projected Sums

For a clause P in projected form, we can assume the constraints are of the form:

$$\exists \vec{\alpha} \text{ s.t. } A\vec{\alpha} \leq \vec{b}$$
$$\wedge\ \vec{s} = Q\vec{\alpha} + \vec{q}\ \wedge\ \vec{v} = R\vec{\alpha} + \vec{r}$$

Here, $\vec{s}$ is a vector of symbolic constants (variables not in V), $\vec{v}$ is a vector of the variables in V, and $\vec{\alpha}$ is a vector of wildcards: quantified variables used only in this clause. We have used matrix notation here because we will utilize some linear algebra theory. This form is the worst-case situation we will need to encounter (generally, only a few or no symbolic variables or variables in V are defined by projections). However, we can easily convert an arbitrary clause into this form.

We calculate the Smith normal form [Sch86] of Q:

$$U \begin{bmatrix} D & 0 \\ 0 & 0 \end{bmatrix} V$$

where U and V are unimodular matrices and D is a integer diagonal matrix (in Smith normal form, D has other properties but these are not important to us here). The constraints defining $\vec{s}$ can be rewritten as

$$U^{-1}(\vec{s} - \vec{q}) = \begin{bmatrix} D & 0 \\ 0 & 0 \end{bmatrix} V\vec{\alpha}$$

We substitute $\vec{\beta}$ for $V\vec{\alpha}$, and partition U^{-1} and β into top and bottom portions to reflect the block structure of $\begin{bmatrix} D & 0 \\ 0 & 0 \end{bmatrix}$. This gives

$$\begin{bmatrix} U_T^{-1} \\ U_B^{-1} \end{bmatrix} (\vec{s} - \vec{q}) = \begin{bmatrix} D & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \vec{\beta}_T \\ \vec{\beta}_B \end{bmatrix}$$

$$\equiv U_T^{-1}(\vec{s}-\vec{q}) = D\vec{\beta}_T \ \wedge \ U_B^{-1}(\vec{s}-\vec{q}) = 0$$

Let $\vec{d}$ be the diagonal of D. These constraints can be rewritten as:

$$d \mid U_T^{-1}(\vec{s}-\vec{q}) \ \wedge \ U_B^{-1}(\vec{s}-\vec{q}) = 0$$

We can now calculate $\vec{\alpha}$ in terms of $\vec{s}$ and $\vec{\beta}_B$:

$$\vec{\alpha} \equiv V^{-1} \begin{bmatrix} D^{-1}U_T^{-1}(\vec{s}-\vec{q}) \\ \vec{\beta}_B \end{bmatrix}$$

In the following equations, we need to syntactly substitute this expression for α. To keep the expressions simple, we do not show the results of this substitution.

Therefore, our constraints P are equivalent to:

$$d \mid U_T^{-1}(\vec{s}-\vec{q}) \ \wedge \ U_B^{-1}(\vec{s}-\vec{q}) = 0$$

$$\wedge \ \exists \vec{\beta}_B \text{ s.t. } (A\vec{\alpha} \leq \vec{b} \ \wedge \ \vec{v} = R\vec{\alpha} + \vec{r})$$

Therefore, $(\Sigma V \ : \ P \ : \ x)$ is equivalent to

$$(\Sigma : d \mid U_T^{-1}(\vec{s}-\vec{q}) \wedge \ U_B^{-1}(\vec{s}-\vec{q}) = 0 :$$

$$(\Sigma \vec{\beta}_B \ : \ A\vec{\alpha} \leq \vec{b} \ : \ x'))$$

where x' is x with $R\vec{\alpha} + \vec{r}$ substituted for $\vec{v}$.

4.6 Approximate simplification

If we are only counting solutions, and are interesting in computing simple upper and lower bounds (as opposed to more complicated but exact answers), we can allow the Omega test to simplify P approximately. The Omega test can produce an answers that are an upper bound or lower bound on the solutions to P. The key place where this approximation occurs is in performing elimination. When an elimination would cause splintering, the Omega test can instead return the real shadow (i.e., allow the eliminated variable to take on real values) or the "dark shadow" (a conservative bound on the solution, described in [Pug92]).

5 Disjoint disjunctive normal form

We now describe techniques that allow us to simplify Presburger formulas so as to produce disjoint disjunctive normal form. We use three different techniques:

- Summarizing uniformly generated sets – In a number of applications, we need to express the memory locations touched by a set of array references (e.g., by the array references `a[i]` and `a[i+1]` in a loop in which `i` runs from 1 to `n`).

  Naively building a Presburger formula to represent the memory locations m touched by this loop:

  $$(\exists i \ : \ 1 \leq i \leq n \wedge m = i) \vee (\exists i \ : \ 1 \leq i \leq n \wedge m = i+1)$$

  will result in an answer with overlapping clauses. By building the formula in a better way:

  $$(\exists i, d \ : \ 1 \leq i \leq n \wedge 0 \leq d \leq 1 \wedge m = i + d)$$

  we avoid this problem.

- Disjoint splintering – When the Omega test projects away a variable, it may need to splinter the problem: describe the result as the union of several problems. In practice, this can often be avoided but we need to be prepared for it. The splinters the Omega test normally generates may be overlapping. We describe here a way to generate splinters that are guaranteed to be disjoint.

- Conversion to disjoint disjunctive normal form – We describe a method for converting an arbitrary formula in disjunctive normal form into disjoint disjunctive normal form.

5.1 Summarizing uniformly generated sets

When computing the number of memory locations or cache lines touched by a set of references in a set of loops, we often have a situation where many of the references differ only in constant parts, as in the SOR example given by [FST91]:

```
for i = 2 to N-1 do
  for j = 2 to N-1 do
    a(i,j) = (2*a(i,j) + a(i-1,j) + a(i+1,j)
                     + a(i,j-1) + a(i,j+1))/6)/6
```

The elements of `a` touched by this loop are:

$$\begin{aligned} &\{[i,j] \mid 2 \leq i,j \leq N-1\} \\ &\quad \cup \{[i-1,j] \mid 2 \leq i,j \leq N-1\} \\ &\quad \cup \{[i+1,j] \mid 2 \leq i,j \leq N-1\} \\ &\quad \cup \{[i,j-1] \mid 2 \leq i,j \leq N-1\} \\ &\quad \cup \{[i,j+1] \mid 2 \leq i,j \leq N-1\} \end{aligned}$$

We can recognize this as a uniformly generated set [GJ88] with offsets $\{(0,0),(-1,0),(1,0),(0,-1),(0,1)\}$. This set can be described exactly as the integer points inside the convex hull of the points. Therefore, we can summarize the elements x, y of a touched by iteration i, j of this loop as:

$$\{[i+\Delta i, j+\Delta j] \quad | \quad -1 \leq \Delta i + \Delta j, \Delta i - \Delta j \leq 1\}$$

and the elements touched by the entire execution of the loop are:

$$\{[i+\Delta i, j+\Delta j] \quad | \quad 2 \leq i,j \leq N-1$$
$$\wedge \quad -1 \leq \Delta i + \Delta j, \Delta i - \Delta j \leq 1\}$$

Using standard techniques, we can simplify this by eliminating Δi and Δj:

$$\{[i,j] \mid 1 \leq i,j \leq N \wedge 3 \leq i+j \leq 2N-1$$
$$\wedge 2 - N \leq i - j \leq N - 2\}$$

5.1.1 Describing a set of constant offsets with linear constraints

In cases such as this, we must convert a set of m constant offsets $p_1, p_2, \ldots p_m$ into a set of linear constraints. We describe two methods:

1. As described by Ancourt [AI91], we can use 0-1 programming methods. Create m new variables $z_1, z_2, \ldots z_m$, and a set of new constraints $C : 0 \leq z_i \leq 1 \wedge 1 = \sum_{i=1}^{m} z_i$. The points can be summarized by $\{[p] : (\exists z_1, z_2, \ldots z_m \ : \ p = \sum_{i=1}^{m} z_i p_i : C)\}$.

   The stencil above can be summarized as:

   $$\{[x, y] : (\exists z_1, z_2, z_3, z_4, z_5 \ : \ x = z_1 - z_2 \wedge y = z_3 - z_4$$

   $$\wedge\ 0 \leq z_1, z_2, z_3, z_4, z_5 \leq 1 \wedge 1 = \sum_{i=1}^{5} z_i)\}$$

2. We can construct the convex hull of the points and check for non-unit strides among the points (e.g., is the first coordinate always odd or the difference of the first two coordinates always a multiple of three).

   The hull and any stride constraints we find are conservative. We next have to check to see if they are exact. One way to do this is to count the number of solutions to the hull and stride constraints, and compare this to the number of points.

The problem with the first technique is that it depends on the constraint system being able to simplify a 0-1 integer programming problem, an iffy proposition at best. We found that although the Omega test can summarize 4-point and 5-point stencils specified this way as a convex region plus stride constraints, it was unable to produce a convex summary for a 9-point stencil. However, the first approach might be able to summarize sets that are missed by the second. Until we have more experience with these techniques in practice, we plan to try both and use whichever give the better result for each case.

5.2 Disjoint splintering when eliminating variables

If $\beta \leq bz$ and $az \leq \alpha$ (where a and b are positive integers), then $a\beta \leq abz \leq b\alpha$. If z is a real variable, $\exists z$ s.t. $a\beta \leq abz \leq b\alpha$ if and only if $a\beta \leq b\alpha$. Fourier variable elimination eliminates a variable z by combining together all pairs of upper and lower bounds on z and adding the resulting constraints to those constraints that do not involve z. This produces a set of constraints that has a solution if and only if there exists a real value of z that satisfies the original set of constraints.

In [Pug92], we showed how to compute the "dark shadow" of a set of constraints: a set of constraints that, if it has solutions, implies the existence of an integer z such that the original set of constraints is satisfied. Of course, not all solutions are contained in the dark shadow.

For example, consider the constraints:

$$\exists \beta \text{ s.t. } 0 \leq 3\beta - \alpha \leq 7 \wedge 1 \leq \alpha - 2\beta \leq 5$$

Using Fourier variable elimination, we find that $3 \leq \alpha \leq 27$ if we allow β to take on non-integer values. The dark shadow of these constraints is $5 \leq \alpha \leq 25$. In fact, this equation has solutions for $\alpha = 3, 5 \leq \alpha \leq 27$ and $\alpha = 29$.

In [Pug92], we gave a method for generating an additional sets of constraints that would contain any solutions not contained in the dark shadow. These "splinters" still contain references to the eliminated variable, but also contain an equality constraint (i.e., are flat). This equality constraint allows us to eliminate the desired variable exactly. For the example given previously, the splinters are:

$$(\exists \beta \ : \ \alpha = 3\beta \wedge 0 \leq 3\beta - \alpha \leq 7 \wedge 1 \leq \alpha - 2\beta \leq 5)$$

$$(\exists \beta \ : \ \alpha + 1 = 3\beta \wedge 0 \leq 3\beta - \alpha \leq 7 \wedge 1 \leq \alpha - 2\beta \leq 5)$$

$$(\exists \beta \ : \ \alpha - 5 = 2\beta \wedge \beta \text{ s.t. } 0 \leq 3\beta - \alpha \leq 7 \wedge 1 \leq \alpha - 2\beta \leq 5)$$

Simplifying these produces:

$$(\exists \gamma \ : \ \alpha = 3\gamma \wedge 1 <= \gamma <= 5)$$

$$(\exists \gamma \ : \ \alpha = 3\gamma - 1 \wedge 2 <= \gamma <= 6)$$

$$(\exists \gamma \ : \ \alpha = 2\gamma + 5 \wedge 5 <= \gamma <= 12)$$

Our goal was to do so in a way that generate as few sets of constraints as possible. Unfortunately for our current situation, the solutions contained in the additional sets of constraints could overlap with each other and with the solutions in the dark shadow. In Figure 1 we give a technique for eliminating integer variables that produces disjoint subproblems. For contrast, we also give our standard algorithm for performing elimination that produces overlapping subproblems. With the new algorithm, the number of subproblems may be larger, but the fact that they are disjoint is much more valuable for our current applications. For this example, the splinters produced are:

$$(\exists \beta \ : \ \alpha = 3 \wedge 0 \leq 3\beta - \alpha \leq 7 \wedge 1 \leq \alpha - 2\beta \leq 5)$$

$$(\exists \beta \ : \ \alpha = 4 \wedge 0 \leq 3\beta - \alpha \leq 7 \wedge 1 \leq \alpha - 2\beta \leq 5)$$

$$(\exists \beta \ : \ \alpha = 26 \wedge \alpha \geq 5 \wedge 0 \leq 3\beta - \alpha \leq 7 \wedge 1 \leq \alpha - 2\beta \leq 5)$$

$$(\exists \beta \ : \ \alpha = 27 \wedge \alpha \geq 5 \wedge 0 \leq 3\beta - \alpha \leq 7 \wedge 1 \leq \alpha - 2\beta \leq 5)$$

Simplifying these produces $\alpha = 3$ and $\alpha = 27$ (the other two clauses simplify to false).

This disjoint splintering is primarily useful only as the last projection step. Consider computing $\exists y, z$ s.t. P, where P is a set of constraints over x, y and z. If neither y nor z can be eliminated exactly without splintering, we have a problem. Say we perform disjoint elimination of z to get:

$$\exists y \text{ s.t. } P_1 + P_2 + \cdots + P_q$$

The problem is that we cannot distribute the $\exists y$ over the disjoint union's without destroying the disjoint property. For example, $1 \le x \le y \le 10$ is disjoint from $1 \le y < x \le 10$, but $\exists y$ s.t. $1 \le x \le y \le 10$ is not disjoint from $\exists y$ s.t. $1 \le y < x \le 10$.

Fortunately, this problem is not too severe. We have found that we frequently do not need to splinter any eliminations. When we do, we often need do only one such elimination and we can postpone it to the very last elimination. If we are forced to perform multiple splintering eliminations, only the last elimination is done with disjoint splinters, and then the techniques described in the next section are used to transform the entire formula into disjoint disjunctive normal form.

5.3 Converting arbitrary DNF formulas into disjoint DNF formulas

Given a formula in disjunctive normal form that may have overlapping conjunctions, we perform the following steps.

Step 1 Check to see if any conjunct is a subset of another conjunct [PW92]. If so, eliminate the one that is a subset.

Step 2 Compute the connected components of the conjunctions, where there is an edge between two conjunctions if they overlap.

Consider each connected component separately in steps 3 and 4.

Step 3 Within each component, pick a conjunction to extract. The selection criteria are:

1. If possible, pick a conjunction that is an articulation point of the graph constructed in step 2.
2. Pick the conjunction with the fewest constraints

Assume the formula being considered is

$$C_1 \vee C_2 \vee \cdots \vee C_p$$

and we extract C_1. Transform the formula to

$$C_1 \ + \ (\neg C_1 \wedge (C_2 \vee \cdots \vee C_p))$$

If C_1 was an articulation point, removal of C_1 will allow us to break up $C_2 \vee \cdots \vee C_p$ into disjoint sections:

$$C_1 + (\neg C_1 \wedge ((C_{1,1} \vee \cdots \vee C_{1,p_1}) + (C_{2,1} \vee \cdots \vee C_{2,p_2}) + \cdots))$$

We distribute the negated C_1 term across the disjoint sections:

$$C_1 + \neg C_1 \wedge (C_{1,1} \vee \cdots \vee C_{1,p_1}) + \neg C_1 \wedge (C_{2,1} \vee \cdots \vee C_{2,p_2}) + \cdots$$

Step 4 To simplify a term

$$\neg C' \wedge (C_1 \vee \cdots \vee C_q)$$

we first replace C' with

$$C'' \equiv \text{gist } C' \text{ given } C_1 \vee \cdots \vee C_q$$

This is valid because

$$\begin{aligned} A \wedge \neg B &\equiv A \wedge \neg(A \wedge B) \\ &\equiv A \wedge \neg(A \wedge (\text{gist } B \text{ given } A)) \\ &\equiv A \wedge \neg(\text{gist } B \text{ given } A) \end{aligned}$$

We calculate

$$\text{gist } C' \text{ given } C_1 \vee \cdots \vee C_q$$

as

$$(\text{gist } C' \text{ given } C_1) \wedge \cdots \wedge (\text{gist } C' \text{ given } C_q)$$

Next, we perform a disjoint negation of C''. If C'' is

$$c_1 \wedge c_2 \wedge c_3 \wedge \cdots$$

The disjoint negation of C'' is

$$\neg c_1 \ + \ c_1 \wedge \neg c_2 \ + \ c_1 \wedge c_2 \wedge \neg c_3 \ + \ \cdots$$

We now distribute the disjoint negation of C'' across $C_1 \vee \cdots \vee C_q$, and reapply the techniques described here to convert it to disjoint DNF.

6 Related work and examples

Nadia Tawbi [Taw91, TF92, Taw94] describes an algorithm for summing a polynomial over a polytope. This is used [TF92] to estimate the execution time of loops and evaluate the load balance of a loop. Tawbi describes techniques roughly equivalent to what we have described in Sections 4.1 – 4.3. For rational bounds, she computes symbolic answers when feasible and computes average values otherwise. She does not describe how to compute upper and lower bounds or split the problem to compute exact answers. The significant differences between our work and hers are the techniques for handling convex sums (ours are an improvement on hers) and general sums (which she does not address).

In Tawbi's algorithm for convex sums,

Eliminate z from C, producing possibly overlapping subproblems
R = False
C' = all constraints from C that do not involve z
$C'' = C$
for each lower bound on z: $\beta \leq bz$
    for each upper bound on z: $az \leq \alpha$
        $C' = C' \wedge a\beta + (a-1)(b-1) \leq b\alpha$
        % Misses $a\beta \leq abz \leq b\alpha < a\beta + (a-1)(b-1)$
        % Misses $\beta \leq bz < \beta + \frac{(a-1)(b-1)}{a}$
    let $a_{\max}$ = max coefficient of z in upper bound on z
    for $i = 0$ to $((a_{\max} - 1)(b-1) - 1)/a_{\max}$ do
        $R = R \vee C \wedge \beta + i = bz$
% C' is the dark shadow
% R contains the splinters
% $C' \vee (\exists$ integer z s.t. $R) \equiv \exists$ integer z s.t. C

Eliminate z from C, producing disjoint subproblems
R = False
if exists constraints $\alpha \leq cz \leq \alpha + b$ such that $b < c - 1$ then
    %need to perform parallel splintering
    C' = False
    for $i = 0$ to b do
        $R = R \vee \exists z$ s.t. $C \wedge \alpha = cz - i$
else
    C' = all constraints from C that do not involve z
    $C'' = C$
    for each lower bound on z: $\beta \leq bz$
        for each upper bound on z: $az \leq \alpha$
            if $C'' \wedge a\beta + (a-1)(b-1) > b\alpha$ is feasible
                $C' = C' \wedge a\beta + (a-1)(b-1) \leq b\alpha$
                % Misses $a\beta \leq abz \leq b\alpha < a\beta + (a-1)(b-1)$
                for $i = 0$ to $(a-1)(b-1) - 1$ do
                    $R = R \vee C'' \wedge a\beta + i = b\alpha$
                $C'' = C'' \wedge a\beta + (a-1)(b-1) \leq b\alpha$
Simplify and check for feasibility each clause in C' and R
% C' is the dark shadow
% R contains the splinters
% $C' \vee (\exists$ integer z s.t. $R) \equiv \exists$ integer z s.t. C

Figure 1: Algorithms for integer variable elimination with overlapping and with disjoint splintering

- the variables in the summation must be eliminated in a predetermined order,
- no attempt is made to eliminate redundant constraints.

Tawbi handles the problem of empty summations by performing an initial polyhedral splitting step, described in [Taw91, Taw94], so that no summation can be empty. Since this splitting step respects the original elimination order, it may split a summation into more pieces than we do.

Example 1 Tawbi [Taw94] gives an example of:

$$\sum_{i=1}^{n} \sum_{j=1}^{i} \sum_{k=j}^{m} 1$$

Her polyhedral splitting technique transforms this to:

$$\sum_{i=1}^{n} \sum_{j=1}^{i} \sum_{k=j}^{m} 1 \qquad \text{if } m \geq n$$

$$\sum_{i=1}^{m} \sum_{j=1}^{i} \sum_{k=j}^{m} 1 + \sum_{i=m-1}^{n} \sum_{j=1}^{m} \sum_{k=j}^{m} 1 \qquad \text{otherwise}$$

The summations can then be computed using standard computer symbolic algebra techniques.

Our techniques work as follows on this example:

$(\Sigma i, j, k \ : \ 1 \leq i \leq n \wedge 1 \leq j \leq i \wedge j \leq k \leq m \ : \ 1)$
Eliminate redundant constraint $1 \leq i$
$= (\Sigma i, j, k \ : \ 1 \leq j \leq i \leq n \wedge j \leq k \leq m \ : \ 1)$
Sum over k (single upper and lower bound)
$= (\Sigma i, j \ : \ 1 \leq j \leq i \leq n \wedge j \leq m \ : \ m - j + 1)$
Sum over i (single upper and lower bound)
$= (\Sigma j \ : \ 1 \leq j \leq n, m \ : \ (n - j + 1)(m - j + 1))$
Splinter upper bounds for j
$= (\Sigma j \ : \ 1 \leq j \leq n \leq m \ : \ (n - j + 1)(m - j + 1))$
$+(\Sigma j \ : \ 1 \leq j \leq m < n \ : \ (n - j + 1)(m - j - 1))$
Sum over j
$= (\Sigma \ : \ 1 \leq n \leq m \ : \ \frac{mn^2}{2} - \frac{n^3}{6} + \frac{nm}{2} + \frac{n}{6})$
$+(\Sigma \ : \ 1 \leq m < n \ : \ \frac{m^2 n}{2} - \frac{m^3}{6} + \frac{nm}{2} + \frac{m}{6})$

In comparing our technique with Tawbi's, we find that our greater flexibility and our ability to eliminate redundant constraints makes our techniques more efficient for many cases (in this example, we only needed to consider 2 terms rather than 3). Also, the techniques described in Sections 4.5 and 5 are a useful contribution above and beyond her work.

Example 2 Mohammad Haghighat and Constantine Polychronopoulos [HP93a, HP93b] describe a method for volume computation, and give two examples. Their first example is:

$$\sum_{i=1}^{n} \sum_{j=3}^{i} \sum_{k=j}^{5} 1$$

Our techniques compute this as:

$(\Sigma i, j, k \ : \ 1 \leq i \leq n \wedge 3 \leq j \leq i \wedge j \leq k \leq 5 \ : \ 1)$
Eliminate redundant constraint $1 \leq i$
$= (\Sigma i, j, k \ : \ 3 \leq j \leq i \leq n \wedge j \leq k \leq 5 \ : \ 1)$
Sum over k
$= (\Sigma i, j \ : \ 3 \leq j \leq i \leq n \wedge j \leq 5 \ : \ 6 - j)$
Sum over i
$= (\Sigma j \ : \ 3 \leq j \leq 5, n \ : \ (n + 1 - j)(6 - j))$
Splinter upper bounds for j
$= (\Sigma j \ : \ 3 \leq j \leq 5 \leq n \ : \ (n + 1 - j)(6 - j))$
$+(\Sigma j \ : \ 3 \leq j \leq n < 5 \ : \ (n + 1 - j)(6 - j))$
Sum over j
$= (\Sigma \ : \ 5 \leq n \ : \ 6n - 16)$

$+(\Sigma\ :\ 3 \le n < 5\ :\ \frac{24-38n+15n^2-n^3}{6})$

If we further recognize that the second summation is only defined at two points ($n = 3$ and $n = 4$), we realize that it can be defined by a first degree polynomial (i.e., a linear term), and find that it is $5n - 12$. This allows us to simplify the above expression to:

$$(\Sigma\ :\ 5 \le n\ :\ 6n - 16) + (\Sigma\ :\ 3 \le n < 5\ :\ 5n - 12)$$

Haghighat and Polychronopoulos [HP93a, HP93b] derive an answer of

$$\mu(\min(n-2,3))(-(\min(n,5))^3+15(\min(n,5))^2-38\min(n,5)+24)/6 \\ +6\max(n-5,0)$$

where $\mu(x)$ is defined to be 1 if x is positive, 0 otherwise. The answer they derive gives the same answers as ours; the form of their answer is quite different because of the min and max expressions they introduce. We have developed a way of introducing min's and max's into the result. Although it sometimes allows us to avoid splitting a summation because of a multiple upper or lower bound, the results tend to be much more complicated. We have decided that in general that it is not worth generating min's and max's.

Example 3 The second example in [HP93a, HP93b]

$$\sum_{i=1}^{2n} \sum_{j=1}^{\min(i,2n-j)} 1$$

is easily handled by our system:

$$\begin{aligned} &(\Sigma i,j\ :\ 1 \le i \le 2n \wedge 1 \le j \le i \wedge i+j \le 2n\ :\ 1) \\ &\qquad \text{Eliminate redundant constraints } 1 \le i \le 2n \\ &= (\Sigma i,j\ :\ 1 \le j \le i \le 2n-j\ :\ 1) \\ &\qquad \text{Sum over } i \\ &= (\Sigma j\ :\ 1 \le j \le n\ :\ 2n-2j+1) \\ &\qquad \text{Sum over } j \\ &= (\Sigma\ :\ 1 \le n\ :\ n^2) \end{aligned}$$

In comparison, Haghighat's and Polychronopoulos' techniques require 9 steps for their first example and 15 steps for their second example. Haghighat and Polychronopoulos [HP93a, HP93b] do not describe their technique in detail. They give a number of rules that can be used in transforming expressions (e.g., $\mu(xy) = \mu(x)\mu(y) + \mu(-x)\mu(-y)$), but do not describe how to decide which rule to apply when. They, like [TF92], assume the summation must be performed in a predetermined order and do not attempt to eliminate redundant constraints.

In comparing our techniques with theirs, we find that ours is fully defined and is much easy to apply for a number of examples (such as Example 2 from [HP93a], which is much harder for their system to analyze).

Example 4 Ferrante, Sarkar and Thrash [FST91] give methods for computing the number of distinct memory locations and cache lines accessed by a loop nest. This information is useful in evaluating cache effectiveness.

The first example they give is calculating the number of distinct memory locations touched by:

```
for i := 1 to 8 do
  for j := 1 to 5 do
    a(6i+9j-7) = a(6i+9j-7) + 5
```

This question can be phrased and answered within our system as follows:

$$\begin{aligned} &(\Sigma x\ :\ (\exists i,j\ :\ 1 \le i \le 8 \wedge 1 \le j \le 5 \wedge x = \\ &\quad 6i+9j-7)\ :\ 1) \\ &\qquad \text{Simplify using omega test} \\ &= (\Sigma x\ :\ x = 8\ :\ 1)\ +\ (\Sigma x\ :\ (\exists \alpha\ :\ 5 \le \alpha \le \\ &\quad 27 \wedge x = 3\alpha - 1)\ :\ 1)\ +\ (\Sigma x\ :\ x = 86\ :\ 1) \\ &= (\Sigma x\ :\ x = 8\ :\ 1) + (\Sigma \alpha\ :\ 5 \le \alpha \le 27\ : \\ &\quad 1) + (\Sigma x\ :\ x = 86\ :\ 1) \\ &= 25 \end{aligned}$$

Example 5 The second example in [FST91] is to calculate the number of memory locations touched in a Successive Over-Relaxation (SOR) code:

```
for i = 2 to N-1 do
  for j = 2 to N-1 do
    a(i,j) = 2*a(i,j) + a(i-1,j) + a(i+1,j)
                      + a(i,j-1) + a(i,j+1)
```

Using techniques described in Section 5.1, we can state and solve this as shown in Figure 2 (to be similar to [FST91], we assume $N = 500$).

To calculate the number of cache lines touched, we need a mapping from array elements to cache lines. A simple mapping [1] is to state that a reference to element $a[i,j]$ of an array references cache line $[(i-1) \div 16, j]$ (where $\div$ stands for integer division). With this mapping, we generate the following answer for the number of cache lines touched by this loop:

$$\begin{aligned} &(\Sigma x,y\ :\ (\exists i,j,\Delta i,\Delta j\ : \\ &\qquad x = (i+\Delta i - 1) \div 16 \wedge y = j + \Delta j \wedge 2 \le i,j \le 499 \\ &\qquad\quad \wedge -1 \le \Delta i + \Delta j, \Delta i - \Delta j \le 1) \\ &\quad : 1) \\ &\qquad \text{Simplify using omega test} \\ &= (\Sigma x,y\ :\ 0 \le x \le 31 \wedge 1 \le y \le 500\ :\ 1) \\ &= 16000 \end{aligned}$$

We can also perform these computations symbolically. We find that the loop touches $(\Sigma\ :\ N \ge 3\ :\ N^2 - 4)$ distinct memory accesses and

$$(\Sigma\ :\ N \ge 3\ :\ N(1 + (N-2) \div 16))$$

$$+(\Sigma\ :\ N \bmod 16 = 1 \wedge N \ge 17\ :\ N - 2)$$

[1] We could also assume more general mappings, in which the cache lines can wrap from one row to another and in which we don't know the alignment of the first element of the array with the cache lines.

$(\Sigma x,y\ :\ (\exists i,j,\Delta i,\Delta j\ :\ x=i+\Delta i\wedge y=j+\Delta j\wedge 2\leq i,j\leq 499\wedge -1\leq \Delta i+\Delta j,\Delta i-\Delta j\leq 1)\ :\ 1)$
Simplify using omega test
$= (\Sigma x,y\ :\ 1\leq x,y\leq 500\wedge 3\leq x+y\leq 999\wedge -498\leq x-y\leq 498\ :\ 1)$
Put in terms of upper and lower bounds on x
$= (\Sigma x,y\ :\ 1,3-y,498+y\leq x\leq 500,999-y,498+y\wedge 1\leq y\leq 500\ :\ 1)$
Splinter upper bounds on x
$= (\Sigma x,y\ :\ 1,3-y,498+y\leq x\leq 500\leq 999-y,498+y\wedge 1\leq y\leq 500\ :\ 1)$
$+(\Sigma x,y\ :\ 1,3-y,498+y\leq x\leq 999-y\leq 498+y\wedge 999-y<500\wedge 1\leq y\leq 500\ :\ 1)$
$+(\Sigma x,y\ :\ 1,3-y,498+y\leq x\leq 498+y<999-y,500\wedge 1\leq y\leq 500\ :\ 1)$
Resimplify
$= (\Sigma x,y\ :\ 2\leq x\leq 499\wedge 1\leq y\leq 500\ :\ 1)$
$+(\Sigma x,y\ :\ x=500\wedge 2\leq y\leq 499\ :\ 1)$
$+(\Sigma x,y\ :\ x=1\wedge 2\leq y\leq 499\ :\ 1)$
$= 249996$

Figure 2: Computation of number of distinct memory locations touched by SOR loop

distinct cache lines.

The method described in [FST91] works well for many simple cases, but:

- cannot handle coupled subscripts or iterations spaces,
- was not originally designed to compute symbolic answers (although it might be adapted),
- often computes a conservative approximation, and
- uses expensive methods to handle the cache lines touched by a set of references (comparing with our methods for summarizing uniformly generated references).

Example 6 We now work through a more elaborate example, which will require us to utilize a number of the techniques we have described. We also mention some additional techniques, not elaborated here, that allow us to further simplify our result.

$(\Sigma i,j\ :\ 1\leq i\wedge j\leq n\wedge 2i\leq 3j\ :\ 1)$
Splinter by considering $3j$ as even or odd
$\equiv (\Sigma i,j\ :\ 2\mid 3j\wedge 1\leq i\wedge j\leq n\wedge 2i\leq 3j\ :\ 1)$
$+(\Sigma i,j\ :\ 2\mid 3j-1\wedge 1\leq i\wedge j\leq n\wedge 2i\leq 3j\ :\ 1)$
Simplify using Omega test
$\equiv (\Sigma i,j\ :\ (\exists\alpha\ :\ j=2\alpha\wedge 1\leq i\wedge 3\alpha\wedge 2\alpha\leq n)\ :\ 1)$
$+(\Sigma i,j\ :\ (\exists\beta\ :\ j=2\beta-1\wedge 1\leq i\leq 3\beta-2\wedge 2\beta\leq n+1)\ :\ 1)$
Deal with projected clauses
$\equiv (\Sigma i,\alpha\ :\ 1\leq i\leq 3\alpha\wedge 2\alpha\leq n\ :\ 1)$
$+(\Sigma i,\beta\ :\ 1\leq i\leq 3\beta-2\wedge 2\beta\leq n+1\ :\ 1)$
Sum over i
$\equiv (\Sigma\alpha\ :\ 1\leq 2\alpha\leq n\ :\ 3\alpha)$
$+(\Sigma\beta\ :\ 1\leq 2\beta\leq n+1\ :\ 3\beta-2)$
Sum over α and β
$\equiv (\Sigma\ :\ 2\leq n\ :\ \frac{3(n-n\bmod 2)(n-n\bmod 2+2)}{8})$
$+(\Sigma\ :\ 1\leq n\ :\ \frac{(n+n\bmod 2)(3n+3(n\bmod 2)-2)}{8})$

If we now do some additional simplification, we can get a even better result. The guard of the first term is identical to the guard of the second term except that it excludes $n=1$. Upon checking, we find the the value of the first clause for $n=1$ is 0, even if we ignore the guard. So we can safely relax the guard of the first clause to $n\geq 1$ and combine the terms:

$$(\Sigma : 1\leq n : \frac{3(n-n \bmod 2)(n-n \bmod 2+2)}{8} + \frac{(n+n \bmod 2)(3n+3(n \bmod 2)-2)}{8})$$

Simplifying this gives:

$$(\Sigma\ :\ 1\leq n\ :\ \frac{3n^2+2n-4(n \bmod 2)^2+3(n \bmod 2)}{4})$$

We can further simplify this by recognizing that $(n \bmod 2)^2 = (n \bmod 2)$:

$$(\Sigma\ :\ 1\leq n\ :\ \frac{3n^2+2n-n \bmod 2}{4})$$

7 Conclusions

We have described methods that are able to count the number of integer solutions to selected free variables of a Presburger formula, or sum a polynomial over all integer solutions of selected free variables of a Presburger formula. This answer can be given symbolically, in terms of symbolic constants (the remaining free variables in the Presburger formula). This ability has many applications in the analysis and transformation of scientific programs.

The techniques we have described are rather elaborate and complicated. This was necessitated by our desire for a method that could handle an arbitrary Presburger formula. This is necessary for applications such as counting distinct memory accesses, cache line accesses and array elements that need to be communicated in a distributed process. For simpler applications requiring only more limited capabilities, such as described by [TF92, HP93a], we make two simple but important observations:

- Summations over several variables should not presume a order in which to perform the summation
- Eliminating redundant constraints is useful

As of March 1994, we have not implemented the complete system described here. As we do so, we will undoubted learn more about efficient techniques for counting solutions and performing summations over Presburger formulas.

8 Acknowledgements

Thanks to Wayne Kelly and Dave Wonnacott for their close readings of this manuscript, and to my entire research team
Wayne Kelly Vadim Maslov Evan Rosser
Tatiana Shpeisman Dave Wonnacott
for their work on the implementation. This work is supported by an NSF PYI grant CCR-9157384 and by a Packard Fellowship.

9 Further Info

The Omega project is exploring the use of advanced constraint technology in analyzing and transforming scientific programs for execution on supercomputers. Among other topics, we are investigating unified frameworks for reordering transformations [KP93b, KP93a], advanced forms of dependence analysis [PW93b, PW93a], and techniques for dealing with polynomial constraints [MP94]. Much of our research is implemented in publicly available implementations, which are being used by other research groups around the world. More info about our research project or software can be obtained via:

- email: `omega@cs.umd.edu`
- anonymous ftp:

  `ftp.cs.umd.edu : pub/omega`

- world wide web:

  `http : //www.cs.umd.edu/projects/omega`

REFERENCES

[AI91] Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. In *Proc. of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, April 1991.

[Bey81] William H. Beyer, editor. *CRC Standard Mathematical Tables.* CRC Press, 1981.

[FST91] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In *Advances in Languages and Compilers for Parallel Processing*, pages 328–343. The MIT Press, 1991.

[GJ88] D. Gannon and W. Jalby. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, pages 587–616, 1988.

[HP93a] M. Haghighat and C. Polychronopoulos. Symbolic analysis: A basis for parallelization, optimization and scheduling of programs. In Utpal Banerjee et al., editor, *Languages and Compilers for Parallel Computing.* Springer-Verlag, August 1993. LNCS vol. 768; proceedings of the Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing.

[HP93b] M. Haghighat and C. Polychronopoulos. Symbolic analysis: A basis for parallelization, optimization and scheduling of programs. Technical Report 1317, CSRD, Univ. of Illinois, August 1993.

[KP93a] Wayne Kelly and William Pugh. Determining schedules based on performance estimation. Technical Report CS-TR-3108, Dept. of Computer Science, University of Maryland, College Park, July 1993. to appear in Parallel Processing Letters (1994).

[KP93b] 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.

[MP94] Vadim Maslov and William Pugh. Simplifying polynomial constraints over integers to make dependence analysis more precise. Technical Report CS-TR-3109.01, Dept. of Computer Science, University of Maryland, College Park, February 1994. Submitted to CONPAR '94.

[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.

[PW93a] 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.

[PW93b] William Pugh and David Wonnacott. Static analysis of upper and lower bounds on dependences and parallelism. *ACM Transactions on Programming Languages and Systems*, 1993. accepted for publication.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, Chichester, Great Britain, 1986.

[Taw91] Nadia Tawbi. *Parallélization Automatique: Estimation des Durées d'Exécution et Allocation Statique de Processeurs*. PhD thesis, Université Pierre et Marie Curie, April 1991.

[Taw94] Nadia Tawbi. Estimation of nested loop execution time by integer arithmetics in convex polyhedra. In *Proc. of the 1994 International Parallel Processing Symposium*, April 1994.

[TF92] Nadia Tawbi and Paul Feautrier. Processor allocation and loop scheduling on multiprocessor computers. In *Proc. of the 1992 International Conference on Supercomputing*, pages 63–71, July 1992.