Efficient Manipulation of Disequalities During Dependence Analysis

Robert Seater and David Wonnacott

Haverford College, Haverford, PA 19041
davew@cs.haverford.edu
http://www.cs.haverford.edu/people/davew/index.html

Abstract. Constraint-based frameworks can provide a foundation for efficient algorithms for analysis and transformation of regular scientific programs. For example, we recently demonstrated that constraint-based analysis of both memory- and value-based array dependences can often be performed in polynomial time. Many of the cases that could not be processed with our polynomial-time algorithm involved negated equality constraints (also known as *disequalities*).

In this report, we review the sources of disequality constraints in array dependence analysis and give an efficient algorithm for manipulating certain disequality constraints. Our approach differs from previous work in that it performs efficient satisfiability tests in the presence of disequalities, rather than deferring satisfiability tests until more constraints are available, performing a potentially exponential transformation, or approximating. We do not (yet) have an implementation of our algorithms, or empirical verification that our test is either fast or useful, but we do provide a polynomial time bound and give our reasons for optimism regarding its applicability.

1 Introduction

Constraint-based frameworks can provide a foundation for efficient algorithms for analysis and transformation of "regular scientific programs" (programs in which the most significant calculations are performed on arrays with simple subscript patterns, enclosed in nested loops). For example, the detection of memory-based array data dependences is equivalent to testing the satisfiability of a conjunction of constraints on integer variables. The individual constraints may be equalities (such as $i = j + 1$), inequalities (such as $1 \leq i \leq N$), and occasionally disequalities (such as $i \neq j$). (For a discussion of the Omega Test's constraint-based approach to both memory-based (aliasing) and value-based (dataflow) dependence analysis, see [1, 2].)

Satisfiability testing of a conjunction of inequality constraints on integer variables ("integer linear programming") is NP-complete [3], and value-based dependence analysis introduces the further complexity of negative constraints. One might not expect that the constraint-based approach to dependence analysis could yield an efficient algorithm, but empirical tests (such as [1]) have found these techniques to be efficient in practice. We recently investigated the reasons for this efficiency [2], and found that most constraints come from a simpler domain for which polynomial-time satisfiability testing is possible.

While it is possible to construct an arbitrarily complicated integer linear programming problem via memory-based dependence analysis [4], almost all of the problems that arise are conjunctions of equality and inequality constraints from the *LI(2)-unit* subdomain. An inequality constraint is said to be in the *LI(2)* subdomain if it can be expressed in the form $ai + bj + c \geq 0$. It is said to be in *LI(2)-unit* if $a, b \in \{-1, 0, 1\}$. The existing Omega Library algorithms [5] perform satisfiability testing of conjunctions of LI(2)-unit inequality and equality constraints in polynomial time.

Negative constraints, even within the LI(2)-unit subdomain, can also cause exponential behavior of the Omega Library. However, almost all of the conjunctions of constraints that are negated during value-based dependence analysis are so redundant (with respect to other constraints) that the Omega Test can replace them with a single inequality (for example, $1 \leq i \leq N \wedge \neg(i = 1 \wedge 1 \leq N)$ will be converted to $1 \leq i \leq N \wedge \neg(i \leq 1)$, and then to $2 \leq i \leq N$). As long as each negated conjunction can be replaced with a single inequality (and the individual constraints are still LI(2)-unit), the Omega Test performs value-based dependence testing in polynomial time.

Many of the cases that could not be processed with our polynomial-time algorithm involved conjunctions of *disequalities* (negated equality constraints). Disequalities can be produced by disequalities in `if` statements, by equality tests in `if-else` statements, during the negation step of value-based analysis, or when an uninterpreted function symbol is used to represent a non-linear term.

Disequalities can be converted into disjunctions of inequalities ($\alpha \neq \beta \Leftrightarrow (\alpha < \beta \vee \alpha > \beta)$). However, when this is followed by conversion to disjunctive normal form, the size of the problem increases exponentially. Lassez and McAloon [6] observed that, for constraints on real variables, disequalities are *independent*. That is, if no one disequality eliminates all solutions, there is no way for a finite number of disequalities to *add up* and together make the system unsatisfiable. Unfortunately, it is in general possible for disequalities to add up for constraint systems with integer variables. We have developed an algorithm to identify disequalities that cannot add up despite our use of integer variables. We call such disequalities *inert*, and use the term *ert* for disequalities that can add up. Satisfiability testing of r inert disequalities can be handled with $2r$ satisfiability tests of conjunctions of inequalities, rather than the 2^r needed for ert disequalities.

This paper is organized as follows: Section 2 provides formal definitions of inert and ert disequalities, and gives the (very simple) algorithm for satisfiability testing. Section 3 gives our inertness test, and Section 4 discusses the impact it would have on the data structures used in the Omega Library. Section 5 gives the reasons why we believe the test would be useful during dependence analysis. Section 6 discusses related work, and Section 7 presents our conclusions.

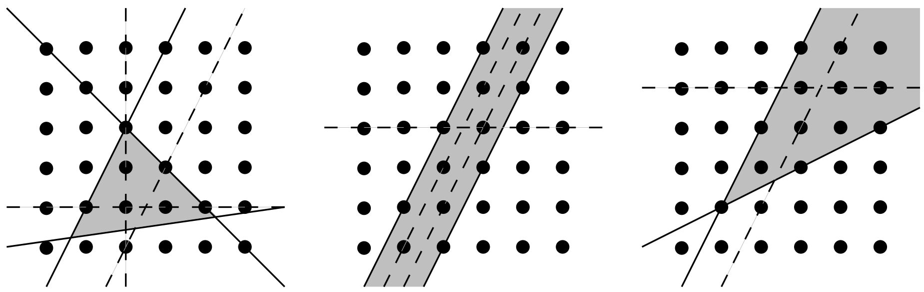
2 Inert (and Ert) Disequality Constraints

From this point on, we consider satisfiability testing of a conjunction of m inequalities and k disequalities on n integer variables. In practice, we may manipulate a mixture of equality, inequality, and disequality constraints, but we ignore equalities here in the interest of simplicity (we could, in principle, convert each equality into a conjunction of inequalities).

Any disequality constraint $\alpha \neq \beta$ can be treated as a disjunction of inequalities $(\alpha < \beta \vee \alpha > \beta)$. However, a satisfiability test of a conjunction of m inequalities and k disequalities using this approach involves 2^k satisfiability tests of conjunctions of $m + k$ inequality constraints (after conversion to disjunctive normal form).

Lassez and McAloon [6] observed that, for constraints on real variables, disequalities are *independent*. That is, if no one disequality eliminates all solutions, there is no way for a finite number of disequalities to add up and together make the system unsatisfiable. Thus, satisfiability testing of a conjunction of m inequalities and k disequalities on real variables can be treated as $2k$ satisfiability tests of $m + 1$ inequalities.

Unfortunately, disequality constraints on integer variables can add up. For example, the three disequalities shown (as dashed lines) in Figure 1a together eliminate all integer solutions in the grey region bounded by the three inequalities (solid lines). In Figure 1b, a collection of four disequalities parallel to the bounding inequalities could eliminate all integer solutions. However, no finite set of disequalities can add up to eliminate all integer solutions in Figure 1c, and disequalities that are not parallel to the bounding inequalities cannot be important in eliminating all integer solutions in Figure 1b. Thus, the opportunity for disequalities to add up depends on the nature of the inequalities and disequalities.



a) No Disequalities can be Inert   b) Some Disequalities are Inert   c) All Disequalities must be Inert

Fig. 1. Inertness of Disequalities on Integer Variables

We therefore give the following definitions:

Given a feasible conjunction of inequalities C and relevant disequality d,

We say d is **inert in C** if, for any finite conjunction of disequalities D, $C \wedge d \wedge D$ is satisfiable $\Leftrightarrow$ both $C \wedge d$ and $C \wedge D$ are satisfiable.

Otherwise, we say that d is **ert in C**.

Note that inertness is not defined when C is infeasible or d is not relevant (in the sense used by Lassez and McAloon [6], i.e. d is relevant if $C \wedge \neg d$ is satisfiable). Our algorithm for satisfiability testing of a conjunction of inequality and disequality constraints follows immediately from this definition:

1. Let C be the inequality constraints and D the disequalities.
2. Test C for satisfiability. If C is unsatisfiable, $C \wedge D$ must be unsatisfiable, so return $false$.
3. Optionally, test each $d \in D$ for relevance (by testing satisfiability of $C \wedge \neg d$), and discard irrelevant disequalities. Note that irrelevant disequalities may be treated as either inert or ert without affecting the result.
4. Test each $d \in D$ for inertness.
5. For each inert disequality $i \in D$, test the satisfiability of $C \wedge i$. If $C \wedge i$ is unsatisfiable, $C \wedge D$ must be unsatisfiable, so return $false$.
6. Let E be the conjunction of all ert disequalities in D. Test $C \wedge E$ for satisfiability by treating each $e \in E$ as a disjunction of inequalities. Return the result of this test (if D contains no ert disequalities, return $true$).

Thus, if we can perform polynomial-time tests for (a) the inertness of a disequality (in Step 4), and (b) the satisfiability of a conjunction of inequalities (in Steps 2, 3, and 5), the overall algorithm is polynomial in the number of inequalities and inert disequalities. The test is still exponential in the presence of ert disequalities (due to Step 6), but we will have reduced the exponent: for i inert disequalities and e ert disequalities, the number of satisfiability tests of conjunctions of inequalities is $2i + 2^e$ rather than 2^{i+e}. Since our existing polynomial-time test requires constraints from LI(2)-unit subdomain, we seek a quick test for inertness within this domain.

Our algorithm for testing inertness is made of two tests that formalize, and generalize for higher dimensions, two insights that are evident from Figure 1. The "closure test" is based on the observation that all disequalities are inert if the set of inequalities is "closed", as in Figure 1a. The "parallel test" finds cases in which parallel inequalities bound an open prism, as in Figure 1b; in this case disequalities that are parallel to the boundaries are ert.

Note that there may be other approaches to polynomial-time satisfiability testing for systems of constraints in the LI(2)-unit subdomain, but our approach follows the philosophy of the Omega Test: produce an algorithm that is efficient in the common cases, but general enough to handle the full logic. This lets us apply a single algorithm to fully LI(2)-unit systems, systems with a few constraints that are slightly more complex, or arbitrarily complex systems of constraints (the last of which may, of course, require unacceptable amounts of memory or time).

3 A Complete Inertness Test for LI(2) Constraints

In this Section we describe a general algorithm for determining inertness of an LI(2) disequality d in a conjunction of m LI(2) inequalities C on n variables. We assume that C is known to have at least on integer solution.

We begin, in Subsection 3.1, by stating a theorem about inertness and outlining an informal proof (so far, our attempts at a full proof have clearly been beyond the scope of this paper). In Subsection 3.2, we give a motivation for the *Closure Test* and an intuitive understanding of how and why it works. In Subsection 3.3 we give detailed pseudocode for the algorithm. In Subsection 3.4, we give and prove upper bounds for the time complexity and space complexity of the algorithm. In Subsection 3.5, we prove the accuracy of the closure test algorithm. In Subsection 3.6, we describe the *Parallel Test* which covers some additional cases which are not accounted for by the closure test. These two tests are the key components to the algorithm to determine inertness. In fact, the proof of correctness (given in 3.7) is just a proof that those tests are sufficient to completely determine inertness.

3.1 Inertness and Non-Parallel Rays

Inertness testing can be viewed as the search for rays contained in C that are not parallel to the hyperplane defined by $\neg d$.

> **Theorem 1.** Given C, a feasible conjunction of inequalities with integer (or rational) coefficients, and d, a disequality relevant to C, d is inert in C iff $\forall$ rays $r \subseteq sol(C), r \parallel d$.

If all rays contained in C are parallel to d, C must be either closed or open in a direction parallel to d, and d is ert in C (recall Figure 1a and the slanted disequalities in Figure 1b).

If C contains a ray r that is not parallel to d, d must be inert in C: Since C is convex, any ray parallel to r with an origin within C contains only points in C. Consider the set of rays R that are parallel to r and originate from the integer solutions to $C \wedge \neg d$. Each ray in R must contain an infinite number of integer points in C (assuming that r is defined, like all our constraints, with integer coefficients). For d to be ert when $C \wedge d$ is satisfiable (which must be the case if C contains a ray not parallel to d), there must be some set of disequalities D for which $C \wedge d \wedge D$ is unsatisfiable and $C \wedge D$ is satisfiable. $C \wedge d \wedge D$ can only be unsatisfiable if every integer solution in C, including those on R, is eliminated by some disequality. Since each ray R_i in R contains an infinite number of integer points, there must be at least one disequality in $(D \wedge d)$ that eliminates all the points in R_i. Since $d \nparallel r$, d cannot eliminate any R_i, and D eliminates all points in R, including all integer solutions to $C \wedge \neg d$. Thus, $C \wedge D$ must be unsatisfiable, and d cannot be ert in C.

3.2 Closure Test: Overview and Motivation

The *Closure Test* determines whether or not each of the variables in d is bounded both above and below. To determine whether or not a variable v is bounded above, we could compute the transitive closure of the "upper bound" relation among the variables (and a single node representing constants). That is, y is an upper bound of x if there exists a constraint $ax \leq by + c$). x is bounded above iff it has a path to the constant node or to both y and $-y$ for some variable y. Recall that in LI(2), there are no bounds of the form $ax \leq by + dz + c$. We actually use a "sloppy" variation on transitive closure that only guarantees accurate bound on the variables in d, to gain a slight reduction in complexity.

It may be tempting to think of determining the boundedness of each variable as equivalent to determining boundedness of C, but that is not the case, and the distinction is important. Any closed region can be trivially made to be open by adding an *irrelevant variable* which is not mentioned in any of the constraints. For instance, adding $z \geq 0$ to C, where z is not mentioned anywhere else in C, will make C open even if it was closed beforehand. However, as far as inertness is concerned, we don't care about the irrelevant variables. This is because d will be extruded infinitely far (without being bounded by C) along each of those variables. Therefore, any ray which "escapes" only along irrelevant variables is necessarily parallel to d. Recall that rays parallel to d do not give us any information about the inertness of d, and that we are only concerned with the existence of non-parallel escaping rays. For these reasons, it is very important that our test treat regions which are only open along irrelevant variables as being closed. In summary, we really want to determine the boundedness of the variables of C which are relevant to d.

3.3 Closure Test: The Algorithm

In this section, we describe the actual algorithm for performing a closure test on a set of variables (namely those in d).

For each variable x in d, we will determine if x is bounded above and if $-x$ is bounded above. Since the lower bound of $x \in d$ is the same as the upper bound of $-x$, we will determine both upper and lower bounds of each variable. However, framing the question entirely as upper bounds will make the algorithm more readable and will make storing and retrieving the information easier.

We will need the following additional storage space to run the algorithm:

a boolean array of length n, recording if a variable has been reached or not.
a boolean table of indirect upper bounds
- one row per variable and per negative of each variable
- one column per variable and per negative of each variable
- one "constant bound" column
- one "modified" column

Pseudocode for the algorithm is as follows:

```
Indirect_Bound(integer x)
  mark x as ''reached''   // prevent infinite recursion
  look at all upper bounds on x and set the appropriate
                        column in the row for x to true
  set the ''modified'' column for x iff any columns were set
  if no constant bound and no +y/-y pair is checked for x
    foreach single variable bound on x which is not ''reached''
      call this Indirect_Bound recursively on it
      set (to true) the column for each variable returned
    // note that there are no multi-variable bounds in LI(2)
  return the list of bounds on x (the true entries in x's row)
```

We run this algorithm on each variable in d, and on the negative of each variable in d. After doing so, we need to do some post-processing:

(1) If any variables in d are entirely unbounded (0 entries in the "modified" column), the the region is unbounded. Otherwise, run the next test.
(2) For each variable in d, check to see if it is either bounded by a constant or bounded by y and $-y$ for some variable y. If each variable is bounded in this manner, then the region is bounded. Otherwise, the region is bounded.

Interpret the results of the algorithm as follows:

If the region is bounded, the d is ert.
If the region is unbounded, then d might still be ert, so we run the Parallel test.

If we were not working over the LI(2) (or LI(2)-unit) domain, then there would also be the possibility of a multiple variable bound.

3.4 Closure Test: Time Complexity

Pre-processing and initializing the table will take $O(n^2)$ time.

Post-processing takes $O(m)$ in order to scan the relevant entries in the table, since the density of the constraints is bounded to 2.

Naively examined, the recursive function will take $O(m^2 n)$ time, however amortized analysis reveals that the test actually takes $O(mn)$ time. That is, by recording our progress in the table, we save a linear amount of time by consulting the table instead of re-deriving some of the information.

The functions is called at most once on each of the n variables. At each call, each of the (up to m) upper bounds has to be examined. Each of those might return as many as m upper bounds which have to be merged with the existing upper bounds. However, the total number of upper bounds is at most m, since each equation only provides one bound on x. The algorithm takes advantage of this fact by not returning previously visited bounds, and thus the total amount of work spent on returning upper bounds in $O(m)$. Thus, the total amount of work done over all n recursive calls is $O(nm)$. Consequently, the overall time complexity is $O(n^2 + mn)$.

3.5 Closure Test: Accuracy

Recall that the quadratic time complexity is achieved because we don't return upper bounds which have already been encountered. However, this means that the bound in question has already been returned to the original variable we are testing, so it is already accounted for. Of course, the upper bounds recorded in the table may be incomplete for the variables which don't appear in d, but we still get accurate information on the variables in d.

Intuition If is x is eventually bounded by y and $-y$, then we can use back substitution to create two constraints of the form

$$\begin{aligned} x + y + c_1 &\geq 0 \\ x - y + c_1 &\leq 0 \end{aligned}$$

Solving for the intersection of those two lines gives us a (constant) bound on x (although not necessarily a tight one).

3.6 The Parallel Test

The *Parallel Test* is based on the fact that if there are two parallel inequalities, one on each side of d, then d will not be inert. For some disequality of the form

$$a_1x_1 + ... + a_nx_n \neq c_0$$

we look for a pair of inequalities of the form

$$\begin{aligned} a_1x_1 + ... + a_nx_n &\leq c_1, \\ a_1x_1 + ... + a_nx_n &\geq c_2 \end{aligned}$$

with $c_2 \leq c_0 \leq c_1$. If such constraints are present in C, then d is ert in C.

The Omega Library uses a hash table to facilitate identification of parallel constraints, so this test should take constant time. Even without this hash table, it would only take $O(m)$ time to scan the m constraints.

Note that outside of the LI(2) subdomain, it is possible to have a case in which d is contained in a "prism" with sides that are not parallel to d. Thus our parallel test is not sufficient to identify all ert disequalities if C includes constraints outside of this subdomain.

3.7 Combining the Two Tests

In this section, we bring together the closure and parallel tests to create a single tests which will completely determine inertness.

> **Conjecture:** Let d and C be LI(2) (or LI(2)-unit). If d is ert in C, then either the closure test or the parallel test will identify it as such. If d is inert, then neither the Closure Test nor the Parallel test will identify it as ert.

That is, the Closure and Parallel tests completely determine inertness. We will validate this conjecture by proving the following theorem.

> **Definition:** Consider a hyperplane (in our case d) and a conjunction of linear constraints C. Let $\boldsymbol{r}$ be a ray which originates on d. If $\boldsymbol{r}$ is completely contained within C but does not intersect the boundary of C, then $\boldsymbol{r}$ is said to **escape C from d**.

For the following theorem and proof, we will use "d" to denote the hyperplane defined by the negation of the disequality d.

> **Theorem 2.** If there are not two non-redundant constraints parallel to d and there exists some ray $\boldsymbol{r}$ such that
> (a) $\boldsymbol{r}$ is not bounded by C,
> (b) the initial point of $\boldsymbol{r}$ satisfies C and lies on d, and
> (c) $\boldsymbol{r} \parallel d$,
>
> then there must exist a ray $\boldsymbol{r'}$ such that
>
> (a) $\boldsymbol{r}$ is not bounded by C,
> (b) the initial point of $\boldsymbol{r}$ satisfies C but not d, and
> (c) $\boldsymbol{r} \nparallel d$.

Remark: The theorem exactly says that d is inert in C only if both tests fail to return "ert". Proving this theorem will also validate the completeness conjecture.

Proof: Since there are not two constraints parallel to d, at least one of the two half spaces defined by d doesn't contain a constraint parallel to d. Consider that half space (or one of them if both fit the criteria). By "down" we will mean directly towards d and by "up" we will mean directly (perpendicularly) away from d. Angles will be implicitly measured from the plane d "upwards".

We will prove that, if all non-parallel rays are blocked by C, then all rays are blocked by C. Thus we will have proven the converse of the theorem and the theorem will follow.

By assumption, there must be some ray, $\boldsymbol{r'}$, which is

(1) parallel to d, and
(2) has initial point on d and within the bounds of C
(3) which is not bounded by C.

If not, then we will construct a valid $\boldsymbol{r'}$ with the following algorithm, beginning it with $n = 0$.

Algorithm: We are given a ray $\boldsymbol{r_n}$. If it is not bounded by C, then stop. We have found a valid $\boldsymbol{r'}$. If so, then there are three ways for $\boldsymbol{r_n}$ to be bounded by (intersect with) a constraint in C.

(a) A constraint that is parallel to d.
(b) A constraint c such that $\boldsymbol{r}_n$ points into d. That is, the angle of $\boldsymbol{r}_n$ up from the projection of $\boldsymbol{r}_n$ onto d is positive.

(c) A constraint c such that $\boldsymbol{r}_n$ does not point into d. That is, the angle of $\boldsymbol{r}_n$ up from the projection of $\boldsymbol{r}_n$ onto d is negative.

The Theorem gives us that (a) is not the case.

If (c) is the case then that constraint must also block $\boldsymbol{r_p}$. This result is a contradiction with our assumption that $\boldsymbol{r_p}$ is not blocked by C and thus cannot occur.

If (b) is the case, then consider a new ray, r_{n+1}, of smaller angle, which is not blocked by the same constraint. Run the algorithm on this new ray. Since there are only a finite number of constraints in C, then eventually a ray will be produced which is blocked by one of the other two cases. Since we are given that case(a) does not occur, we know that eventually we will produce a ray which is not bounded by C.

Consequently, if both the Parallel Test and the Closure Test do not return "ert", then a non-parallel ray must escape – making d inert in C.

3.8 Generalizations

The algorithms above work when all inequalities and disequalities are in the LI(2) subdomain. If a formula contains a small number of disequalities outside of this subdomain, we can safely (if expensively) treat them as ert. However, if a formula contains one non-LI(2) inequality, we must (in the absence of an inertness test) treat *all* disequalities as ert.

We are currently investigating extensions of our inertness test, focusing on the use of linear programming techniques to perform a direct test for the existence of a ray r that is not parallel to d: we simply determine if an objective function that is perpendicular to d is unbounded. However, it is not clear that the overall complexity of this approach can be made low enough to make it helpful in dependence testing.

We could also try to identify the extreme rays of C, and then determine whether or not any of them are not parallel to d. However, if a system has many extreme rays and few disequalities, this might prove to be much slower than our approach.

It is also worth noting that any disequality that contains a variable that does not appear in any inequality (or equality) constraint can trivially be satisfied (essentially by treating the disequality as an equality, solving for the new variable, and then setting it to some other value).

4 The Representation of "Simplified" Relations

As the Omega Test and Omega Library [5] served as the foundation for our prior work on polynomial time array dependence analysis, it is the obvious framework for implementation of the algorithms presented here. Such an implementation would involve major modifications to the Omega Library's core data structures. The library is designed to transform relations defined by arbitrary Presburger

Formulas [7], possibly with certain uses of uninterpreted function symbols [8], into a “simplified form”. This transformation happens automatically during satisfiability testing and at other times; it prevents redundant analysis, and thus presumably provides a great speed advantage over a system that evaluates every query based on an unsimplified relation.

The simplified form is a variant of disjunctive normal form in which individual “conjuncts” (conjunctions of equality and inequality constraints, possibly with local existentially quantified variables) are connected by disjunction ($\vee$). Depending on the query performed, this simplified form may or may not include redundant conjuncts, equalities, or inequalities.

Note that simplification may not always be beneficial, and deferring it to the proper point is an important strategy for getting good performance from the Omega Library. For example, consider queries for value-based dependence analysis, which have the form $C_0 \wedge \neg C_1 \wedge \neg C_2 \wedge ... \wedge \neg C_N$, where the C_i's are conjuncts. The Omega Library uses information in C_0 to reduce the cost of negating the other conjuncts. If we were to simplify each negated conjunct and then combine the results with $\wedge$, the cost would be dramatically higher for many cases (see [1] for details).

Even if our polynomial-time disequality algorithm has proven that a system of constraints is satisfiable, converting it into simplified form can increase its size exponentially, since each non-redundant disequality will be converted into a disjunction. We could solve this problem by allowing disequality constraints within the individual conjunct data structures. This approach would have benefits even if all disequalities where ert: except in cases where redundancy is to be removed, the Omega Library could stop testing for satisfiability as soon as it has proven a relation is satisfiable. The current algorithms produce the entire disjunction and then test each conjunct for satisfiability. This could provide some part of the speedup shown under “privatization analysis” in [9, Table 13.2], but in a more generally applicable context.

An equivalent approach would be to simply allow negated equality constraints in simplified relations. This approach could be taken even further, to allow more general negated constraints, or other formulas that cannot be handled efficiently (or at all). The current Omega Library can (in principle) handle arbitrary Presburger Formulas when it is not restricted to our provably polynomial subdomain. However, when faced with certain uses of uninterpreted function symbols, or when restricted to provably polynomial cases, the Omega Library replaces any set of constraints that it cannot handle with a special constraint identified simply as *unknown*.

It might be possible to modify this algorithm to annotate each *unknown* with the unsimplified formula that produced it, in case later manipulation of the relation provides information that lets the library handle the offending constraint. However, without extensive empirical testing, it is hard to know whether the overhead involved in this approach would be worthwhile.

The above changes have the potential to improve the accuracy, speed, and ease of use of the Omega Library, since polynomial-time simplifications could be performed early without causing a decrease in later accuracy (this approach would also make the efficiency less sensitive to the timing of simplifications).

5 Implementation Status and Future Work

We do not currently have an implementation of our algorithms, and thus we do not have empirical verification that they are either fast or effective in practice. Given the nature of the changes discussed in the previous section, we do not expect to have an implementation any time soon.

However, we do have reason to hope that our algorithms will be applicable during dependence analysis. Our studies of the constraints that arise in practice [1, 2] suggest that disequalities often involve loop index variables used in `if` statements or in subscripts. For programs with scalable parallelism, some or all loops are bounded by symbolic constants (typically program parameters), which are not themselves bounded above. In this case, we expect the disequalities to be inert. When all disequalities are inert and the constraints obey the other conditions given in [2], memory- and value-based dependence testing can be done in polynomial time.

Before undertaking any implementation effort, we plan to investigate algorithms for projection and gist in the presence of disequalities. It may be the case that some of the insights of Imbert [10] can be combined with our definition of inertness in some useful way.

6 Related Work

Most other work on handling negated constraints during dependence analysis focuses on producing approximate results or deferring satisfiability tests until more constraints are available. The Omega Library's negation algorithms [11, 9] and the algorithms for manipulating "Guarded Array Regions With Disjunction" (GARWD's) in the Panorama compiler [12] are examples of the deferral approach (the proposals at the end of Section 4 were directly inspired by the GARWD algorithms). The drawback with deferring negation is, of course, that we will be forced to choose some other approach if we do not get any helpful constraints before we must answer a satisfiability query.

Our work with identifying inert disequalities complements this approach, and there should be no problem with combining the two. When disequalities are inert, they can be tested directly; when they are not, satisfiability testing should be delayed as long as possible.

We do not know of any other work on polynomial-time satisfiability testing of disequalities on integer variables. Our work on identifying inert disequalities on integer variables was driven by a frustrated desire to apply the work of Lassez and McAloon [6], which is relevant only to real (or rational) variables.

7 Conclusions

Disequality constraints can cause exponential behavior during dependence analysis, even when all constraints are in the otherwise polynomial LI(2)-unit domain. We have developed a polynomial-time algorithm to identify certain inert disequalities within this domain, in which case satisfiability testing is polynomial in the number of inequalities and inert disequalities, but exponential in the number of ert disequalities.

The integration of our algorithms into the Omega Library would require a redefinition of the central data structure representing a "simplified" problem, and would thus be a major undertaking. However, it might provide opportunities for improving the speed and accuracy with which the Omega Test handles other queries.

8 Acknowledgments

This work is supported by NSF grant CCR-9808694.

References

1. William Pugh and David Wonnacott. Constraint-based array dependence analysis. *ACM Trans. on Programming Languages and Systems*, 20(3):635–678, May 1998.
2. Robert Seater and David Wonnacott. Polynomial time array dataflow analysis. In *Proceedings of the 14th International Workshop on Languages and Compilers for Parallel Computing*, August 2001.
3. Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.
4. D. E. Maydan, J. L. Hennessy, and M. S. Lam. Efficient and exact data dependence analysis. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 1–14, June 1991.
5. Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. The Omega Library interface guide. Technical Report CS-TR-3445, Dept. of Computer Science, University of Maryland, College Park, March 1995. The Omega library is available from http://www.cs.umd.edu/projects/omega.
6. Jean-Louis Lassez and Ken McAloon. Independence of negative constraints. In *TAPSOFT 89: Proceedings of the International Joint Conference on Theory and Practice of Software*, 1989.
7. G. Kreisel and J. L. Krevine. *Elements of Mathematical Logic*. North-Holland Pub. Co., 1967.
8. Robert E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the ACM*, 26(2):351–360, April 1979.
9. David G. Wonnacott. *Constraint-Based Array Dependence Analysis*. PhD thesis, Dept. of Computer Science, The University of Maryland, August 1995. Available as `ftp://ftp.cs.umd.edu/pub/omega/davewThesis/davewThesis.ps`.
10. Jean-Louis Imbert. Variable elimination for disequations in generalized linear constraint systems. *The Computer Journal*, 36(5):473–484, 1993. Special Issue on Variable Elimination.

11. William Pugh and David Wonnacott. An exact method for analysis of value-based array data dependences. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, August 1993.
12. Junjie Gu, Zhiyuan Li, and Gyungho Lee. Experience with efficient array data flow analysis for array privatization. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 157–167, Las Vegas, Nevada, June 1997.