Haverford College Computer Science Tech Report 2010-01
Archived 19 June 2010

A Retrospective of the Omega Project

David G. Wonnacott

davew@cs.haverford.edu

Haverford College
Haverford, PA 19041

Abstract

The Omega Project, led by Bill Pugh of the University of Maryland at College Park, explored advanced techniques for analysis and transformation of dense numeric codes. This project was one of the first to use affine constraints on integer variables for static program optimization. In addition to numerous articles, the Omega Project released source code for many of their algorithms. Most of their constraint manipulation algorithms were released as the Omega Library and the text-based Omega Calculator; many of their program transformation algorithms were released as Petit ("Pugh's Extended Tiny Isn't Tiny"), a tool for exploring static program transformation that was based on Michael Wolfe's "tiny" system.

This document gives a brief description of the major results of the Omega Project, including both approaches to program optimization and algorithms for constraint manipulation.

1 Exact Instance-Based Array Dependence Information

Optimization of loop nests requires information about dependences that is both detailed and accurate. Even for the restricted case in which array subscripts and loop bounds must be affine functions of loop indices, testing for the existence of a dependence is equivalent to testing satisfiability of a conjunction of affine constraints on integer variables, which is NP-complete [GJ79]. Fortunately, in practice most array subscripts and loop bounds are even simpler, and low-complexity tests such as the GCD test and Banerjee's Inequalities produce accurate results in many cases.

In the late 1980's and '90's, Paul Feautrier and Bill Pugh led independent efforts to improve array dependence analysis with techniques that were exact for the entire affine domain and fast for the common cases [Fea88, Pug91a, Pug92]. While some researchers felt exponential algorithms had no place in compilers, others saw value in the more precise information produced by these techniques. This approach, known variously as "constraint-based analysis", the "polyhedral model", or "instance-wise analysis", remains an important tool for advanced restructuring compilers, some of which make use of the original code of Pugh or Feautrier. The remainder of this article reviews the work of Pugh's Omega Project; information about other work on this area must be found elsewhere, such as the upcoming "Encyclopedia of Parallel Computing".

1.1 Traditional Dependence Abstractions

To illustrate some of the issues involved in dependence analysis, consider the codes shown in Figure 1 — each of these loops might write an element of array A that is read in a later iteration, so each is said to exhibit an inter-iteration flow dependence based on A, which inhibits parallel execution of the loop iterations. Earlier work on dependence analysis not only approximated some affine cases, but also reported only simple dependence abstractions, e.g., identifying loops that carry a dependence or classifying dependences with direction vectors or distance vectors. Regardless of its accuracy or domain, any test that produces this information must report that both loops carry a dependence with a distance that is not known at compile time (assuming we lack information about n and k).

A more detailed examination of the dependences of Figure 1 reveals that the loops can be parallelized with different strategies, as illustrated in Figure 2 (circles correspond to iterations, with lower values of i to the left, and each arrow indicates a flow dependence between iterations). For the illustrated case ($n = 8$ and $k = 3$), Figure 1a has dependences from iteration 0 (which writes a value into A[0]) to iteration 7 (which reads a value from A[0]), from 1 to 6, from 2 to 5, and from 3 to 4; Figure 1b has dependences from 0 to 3, 1 to 4, 2 to 5, 3 to 6, and 4 to 7. These patterns suggest that there may be some cases in which we can run some of the iterations in parallel, e.g., when $n = 8$ and $k = 3$ we could execute iterations 0-3 of Figure 1a simultaneously, and iterations 0-2 of Figure 1b simultaneously. However, conclusions about program transformation must be made about the general case.

1.2 Dependence Relations

The Omega Test describes the dependences in terms of relations among tuples of integers. To retain correspondence with the program being analyzed, these integers may be identified as inputs (e.g., the source of a dependence), outputs (the sink), or symbolic constants. A finite set of dependences can be given as a list (union) of such tuple relations, e.g., for Figure 2a as $\{[0] \rightarrow [7]\} \cup \{[1] \rightarrow [6]\} \cup \{[2] \rightarrow [5]\} \cup \{[3] \rightarrow [4]\}$. Integer variables and constraints allow a representation that is both concise (allowing $\{[i] \rightarrow [7 - i] \mid 0 \leqslant i \leqslant 3\}$ for Figure 2a) and general ($\{[i] \rightarrow [n - 1 - i] \mid 0 \leqslant i < \frac{n}{2}\}$ when n is not known, as in Figure 1a).

```
// Example 1a                    // Example 1b
for (i=0; i<n; i++)              for (i=0; i<n; i++)
   A[i] = A[n-1-i]*B[i];            A[i+k] = A[i]*B[i];
```

Figure 1. Simple Loops Exhibiting Data Dependences

Figure 2. Iteration Spaces and Flow Dependences for Figure 1, when $n=8$ and $k=3$.

These dependence relations follow directly from the application of the definition of data dependence to the program to be analyzed. A flow dependence exists between a write and a read (e.g., from iteration $[i]$ to $[i']$) when the subscript expressions are equal (e.g., $i = n-1-i'$ in 1a), the loop index variables are in bounds ($0 \leqslant i < n \wedge 0 \leqslant i' < n$), and the dependence source precedes the sink ($i < i'$). Combining these constraints produces $\{\,[i] \rightarrow [i'] \,|\, i = n-1-i' \wedge 0 \leqslant i < n \wedge 0 \leqslant i' < n \wedge i < i'\,\}$ for 1a and $\{\,[i] \rightarrow [i'] \,|\, i' = i+k \wedge 0 \leqslant i < n \wedge 0 \leqslant i' < n \wedge i < i'\,\}$ for 1b. These dependence relations are **exact** in the following sense: for any given values of the symbolic constants and loop indices, the constraints evaluate to true if and only if a dependence would actually exist. The constraint-manipulation algorithms of the Omega Library (see Section 3) can confirm that these relations are satisfiable and produce a simpler form, in these cases $\{\,[i] \rightarrow [n-1-i] \,|\, 0 \leqslant i \wedge 2i \leqslant n-2\,\}$ ($2i \leqslant n-2$ is equivalent to $i < \frac{n}{2}$ but does not introduce division) and $\{\,[i] \rightarrow [i+k] \,|\, 0 \leqslant i < n-k\,\}$.

This representation of dependences extends naturally to nested loops with the introduction of additional variables in the input and output tuple — dependences among references nested in loops i and j would be represented as relations from $[i, j]$ to $[i', j']$. An imperfect loop nest such as that shown in Figure 3 raises the additional challenge of identifying the lexical position of each

```
for (t = 0; t < T; t++) {
  for (i = 1; i < N-1; i++) {
    new[i] = (A[i-1] + A[i] + A[i+1]) * (1.0/3);
  }
  for (i = 1; i < N-1; i++) {
    A[i] = new[i];
  }
}
```

Figure 3. A Set of "Imperfectly Nested" Loops

statement or inner loop (i.e. is it the first, second, etc. component within the loop?). This challenge can be met by interspersing (among the loop index values) constant values that give the statement position. If the i loop in Figure 1 is the first (or only) statement in its function, and the update to A is the first (or only) statement in this loop, we could represent the dependence above as a relation from $[1, i, 1] \rightarrow [1, i', 1]$. For Figure 3, the dependence from the write of A[i] to the read A[i-1] is $\{\,[1, t, 2, i, 1] \rightarrow [1, t', 1, i', 1] \,|\, i'-1 = i \wedge t < t' \wedge 0 \leqslant t, t' < T \wedge 1 \leqslant i, i' < N-1\,\}$.

While it is possible to produce dependence distance/direction vectors from the dependence relations produced by the Omega Test, the results are often no more accurate than those produced by earlier tests such as a combination of Banerjee's inequalities and the GCD test. However, the added precision of the dependence relation can enable other forms of analysis or transformation if it is passed directly to later steps of the compiler.

2 Advanced Program Analysis and Transformation

The detailed information present in a dependence relation can be used as a basis for a variety of program analyses, including analysis of the flow of values within a program's arrays, analysis of conditional dependences, and analysis of dependences that exist over only a subset of the iteration space. This information can then be used to drive a number of program transformation/optimization techniques on perfectly or imperfectly nested loops.

2.1 Advanced Analysis Techniques: The Omega Test

Information about the iterations involved in a dependence can reveal opportunities for parallelization after transformations such as index set splitting or loop peeling. Note that any iterations that are not the source of a dependence arc (e.g., iterations 4-7 of Figure 2a) can be executed concurrently after all other loop iterations have been completed; and those that are not a sink (iterations 0-3 in Figure 2a) can be executed concurrently before any other has started. The Omega Library's ability to represent sets of integer tuples as well as relations can be used to show that there are never any iterations other than these non-sink and non-source iterations for Figure 1a, so this loop (unlike that in 1b) can be run in two sequential phases of parallel iterations.

The symbolic information present in a dependence relation can be used to detect conditional dependences. The flow dependence relation for Figure 1b clearly implies that the flow dependence exists only when $0 < k < n$; the Omega Library's projection and "gist" operations (see Section 3.2) can be used to extract this information. A compiler could introduce conditional parallelism (e.g., run the loop in parallel when $n \leqslant k$) or use additional analysis or queries to the programmer to attempt to prove no dependence ever exists.

By combining instance-wise information about memory aliasing with information about iterations that overwrite array elements, the Omega Test can produce information about the flow of values (termed "value-based dependence information" by the Omega Project). For example, the "memory-based" flow dependence shown above for Figure 3 connects writes to reads in all subsequent iterations of the t loop — i.e., when $t' > t$. However, the value written in iteration t is overwritten in iteration $t + 1$, and thus never reaches any subsequent iteration. The value-based dependence relation includes this information, showing dependences only when $t' = t + 1$, as shown in Section 3.3. Value-based dependence information can be used for a variety of optimizations, e.g. to identify opportunities for parallelization after array privatization or expansion.

The Omega Project uses the term "Omega Test" for the production of symbolic instance-wise memory- or value-based dependence relations via the steps outlined above. More detail about the specific algorithms involved in dependence testing can be found in [PW98] as well as earlier publications by these authors.

2.2 Iteration Space Transformation

Iteration space transformations can be viewed as relations between elements of the original and transformed iteration space, and thus represented and manipulated with the same infrastructure as dependence relations. Iteration spaces can be related to execution order via a simple rule such as "iterations are executed in lexicographical order", thereby turning iteration space transformation into a tool for transforming order of execution. A similar approach can be used to describe and transform the association of values to memory cells [Won02], though this has not been explored as thoroughly.

The "transformation as relation" framework unifies all unimodular loop transformations (e.g., $\{[i,j] \rightarrow [j,i]\}$ describes loop interchange) with a number of other transformations. Perhaps more importantly, this framework handles imperfect loop nests, which lie outside the domain of unimodular techniques. For example, consider the problem of fusing the inner loops of Figure 3. Simple loop fusion can be accomplished by making A[i] = new[i] into the *second* statement in the *first* i loop, i.e., $\{[1, t, 2, i, 1] \rightarrow [1, t, 1, i, 2]\}$. However, fusion without first aligning the loops is illegal; $\{[1, t, 2, i, 1] \rightarrow [1, t, 1, i + 1, 2]\}$ is a correct alignment and fusion (now element A[1] is overwritten in iteration i=2, just after its last use in the computation of new[2]). This transformation produces a loop nest with much better memory locality, since (barring interference) it moves each array into cache only once per time step rather than twice. Algorithms for finding useful program transformations in this framework, and for generating code from the resulting iteration space sets, are discussed in [Pug91b, KP94, KP96, Won00].

2.3 Iteration Space Slicing

Iteration space transformations can also be defined in terms of "iteration-space slicing": the instance-wise analog of program slicing. This approach can express very general approaches to program transformation in terms that are relatively concise and clear.

To illustrate iteration space slicing, consider the challenge of parallelizing the code in Figure 1b even when the dependence exists, as in Figure 2b. Figure 4 shows the iteration space of this

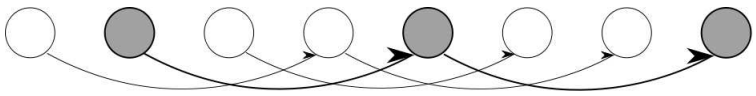


Figure 4. The Backward Iteration Space Slice for iteration 7 of Figure 2b.

loop when $n = 8$ and $k = 3$ (as in Figure 2b), together with the backward iteration space slice for iteration 7 (the set of other iterations that must be executed to produce the value in the target iteration). A similar backward slice for iteration 6 would include iterations 3 and 0.

It is possible to run Figure 1b as k concurrent threads, each of which updates every k^{th} element. A simple test could be used to identify and parallelize this specific case (a one-dimensional loop with a single constant-distance dependence), but iteration space slicing provides a concise way to generalize this idea: find the backward slices to the iterations that are not the source of any dependence — if these slices do not intersect, they can be run concurrently to produce the result of the loop nest.

In some cases, it may be necessary to find the iteration space slice needed to compute one result given that another slice has already been completed. For example, consider the challenge of increasing the memory locality of Figure 3 by more than the factor of two achieved in Section 2.2. Figure 5 shows the iteration space of a sample run of the original form of this code: each time step is shown as a column of circles representing the computations of new[i], followed by a column of squares representing the copy into A[i]. The slice needed to compute A[1] is shown as dashed iterations surrounded by a dashed border; the marginal slice needed for A[2] given that A[1] has been computed is shown as shaded iterations. Executing this code as a series of sequential slices can greatly improve the memory locality (given a cache large enough to hold the results of a few slices, this can reduce memory traffic from $O(T \cdot N)$ to $O(N)$ in the absence of interference, producing dramatic speedups for slow memory systems [Won02]).

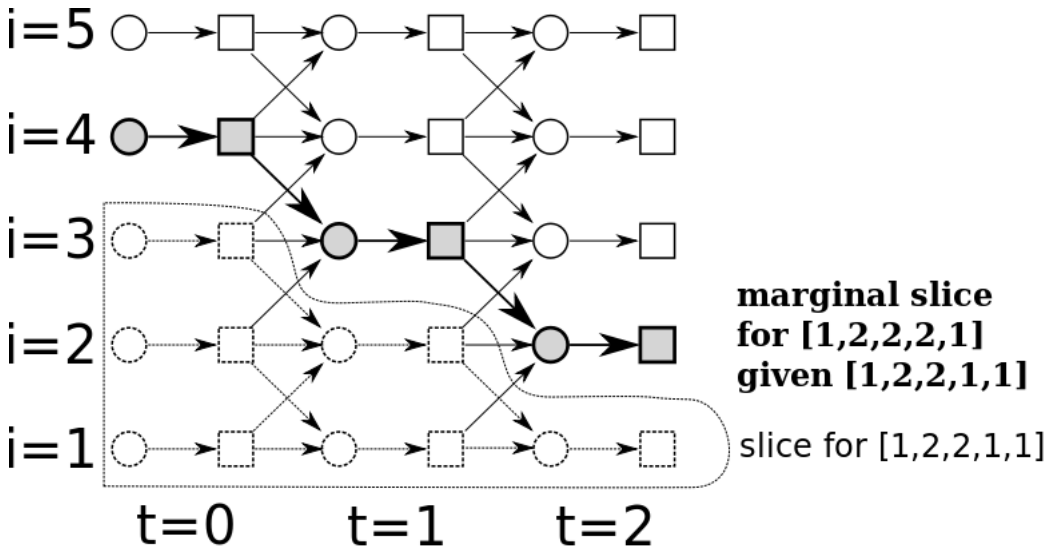


Figure 5. Iteration Space and Two Slices for Figure 3, $T = 3$, $N = 7$.

Algorithms for iteration space slicing, for producing transformations with it, and for generating code to execute slices are described in [PR97, PR99, Won02].

3 Representation and Manipulation of Sets and Relations

The Omega Library (or simply "Omega" when in context) employs a number of techniques to transform the constraints extracted from programs in Section 1 into answers to the questions of Section 2. The details of representation and manipulation depend on the nature of the constraints and the question being asked.

Most algorithms can be expressed either in terms of individual (in)equations or higher order operations on relations and/or sets. For example, a compiler could construct each memory-based flow dependence relation by creating appropriate tuples for input and output variables and then examining loop bounds, subscript expressions, etc., to produce the appropriate constraints on the tuples' variables. Alternatively, it could first define a set for the iteration space I of each statement S (I_s), and a relation M mapping loop iteration to array index for each array reference R (M_R), and relations T for forward-in-time order for various loop depths, and then construct a flow dependence by combining these, i.e. the dependence from access x to y could be found by taking $(M_x \bullet (M_y)^{-1}) \cap T_{xy}$ and restricting its domain to I_x and its range to I_y.

Some operations on relations or sets simply involve re-labeling of free variables, e.g. swapping the input and output tuples to find R^{-1} from a relation R. Others such as intersection ($\cap$) involve matching up variables from input and output tuples, and possibly converting free variables to quantified variables; in the relational join and composition operations (for $R_1 \bullet R_2$ or the equivalent $R_2 \circ R_1$), R_1's input tuple becomes the input tuple, R_2's output tuple the output tuple, and R_1's outputs and R_2's inputs become existentially quantified variables, all constrained to the intersection of the constraints from R_1 and R_2. Finally, some operations such as transitive closure are defined in Omega only as relational operators.

The remainder of this section gives an overview of the algorithms used in the Omega Library. It progresses from less-expressive to more-expressive constraint languages, roughly following the historical development of the algorithms of the Omega Library, and gives descriptions in terms of collections of (in)equations rather than relations or sets. For more detail on these algorithms or the relationship between high and low-level operations, see the cited papers, or [KPRS95] for transitive closure algorithms.

3.1 Conjunctions of Affine Equations and Inequations

The fundamental data structure of the Omega Library represents a conjunction of affine equations and inequations with integer coefficients (Omega does not currently represent disequations such as $n \neq m$, so hereafter the term inequation refers only to $<$, $\leqslant$, $\geqslant$, and $>$ constraints, each of which is converted to $\geqslant$ form internally). Each individual (in)equation is immediately reduced to lowest terms (e.g. turning $3x + 9y + 21 = 0$ into $x + 3y + 7 = 0$), and hash keys that ignore constant terms are used to quickly detect (and remove) simple redundancies (e.g., $a - 2b \geqslant 0$ and $a - 2b + 10 \geqslant 0$ have the same hash key and the latter can be removed). The hashing system also detects (and replaces) pairs of inequations that are equivalent to an equation (e.g. $i \geqslant 5$ and $i \leqslant 5$ are converted to $i = 5$).

High-level operations for these conjunctions rely on lower-level operations on sets of (in)equations, such as variable elimination and redundancy detection. Variable elimination is also referred to as "projection" or finding the "shadow" of a set or relation, since finding the two-dimensional shadow of a three-dimensional object can be thought of as an example of this operation. Note, however, that the shadow of the integer points in a set/relation is *not* always the same as the integer points in the shadow (as per the discussion of Figure 6 below).

To illustrate Omega's variable elimination and redundancy detection abilities, recall the dependence from Figure 1b. Checking for possible dependence corresponds to existentially quantifying a dependence relation's free variables (inputs, outputs, and symbolic constants), in this example checking the formula $(\exists i, i', n, k : i' = i + k \wedge 0 \leqslant i < n \wedge 0 \leqslant i' < n \wedge i < i')$. None of the six (in)equations of this formula is redundant with respect to any one other, so all are retained to the start of our query. Omega begins by using equations to eliminate variables: given $i' = i + k$ it could replace all occurrences of i' with $i + k$, producing the formula $(\exists i, n, k\colon 0 \leqslant i < n \wedge 0 \leqslant i + k < n \wedge 0 < k)$. (Additional properties of integer arithmetic are used to eliminate an equation even when no variable has a unit coefficient, as discussed in [Pug91a].)

Upon running out of equations, Omega moves on to eliminate variables involved in inequations. Inequations are removed in several passes, with quick passes occurring first. Omega checks for variables that are bounded only on one side (i.e., have no lower bounds or no upper bounds). Such variables cannot affect the satisfiability of the system, so they can be removed, along with all constraints on them — in this example, n, then k, and then i are removed in turn, producing an empty conjunction, i.e. True.

To illustrate some of the other algorithms used for affine conjunctions, we consider what would happen in this example if we knew $n \leqslant 1000$ (perhaps the loop in Figure 1 is guarded by the test `if n <= 1000`), i.e., $(\exists i, n, k\colon 0 \leqslant i < n \wedge 0 \leqslant i + k < n \wedge 0 < k \wedge n \leqslant 1000)$. In the absence of variables bounded on only one side, Omega examines groups of three inequalities to determine if any two contradict a third or make it redundant. In our running example, $i + k < n$ and $0 < k$ make $i < n$ redundant, and $0 \leqslant i$ and $0 < k$ make $0 \leqslant i + k$ redundant, reducing the query to $(\exists i, n, k\colon 0 \leqslant i \wedge i + k < n \wedge 0 < k \wedge n \leqslant 1000)$.

Omega then moves on to its most general technique: an adaptation of Fourier's method of variable elimination. Fourier's method can be used to eliminate a variable v in a conjunction of affine inequations, by replacing the set of inequations on v with the set of all possible combinations of one upper and one lower bound on v — for example, replacing $0 \leqslant i$ and $i < n - k$ with $0 < n - k$. The original conjunction has a rational solution if and only if the new one does; the new system has one variable fewer, but may have many more inequations.

The Omega Library uses an extension of Fourier's method that preserves the presence of *integer* solutions. This process eliminates a variable by comparing the result of Fourier's original technique (which produces the "shadow") with a variant Pugh calls a "dark shadow" — a lower-dimensional conjunction has integer solutions *only if* the original did. The set of integer solutions to the lower-dimensional system is the union of this dark shadow and the "splinter" solutions that lie within the shadow but outside the dark shadow [Pug91a].

Omega repeatedly eliminates variables to produce a trivially-testable system with at most one variable, possibly producing $(\exists n, k\colon 0 < n - k \wedge 0 < k \wedge n \leqslant 1000)$ and then $(\exists n\colon 0 < n \leqslant 1000)$ in our running example. Since the query is now clearly satisfiable, the Omega Test concludes that there must be some values of the constants n and k for which a dependence exists between some iterations.

The example above, like most occurring during dependence analysis, does not illustrate the use of splintering to project integer solutions. Figure 6 shows an example for the inequations

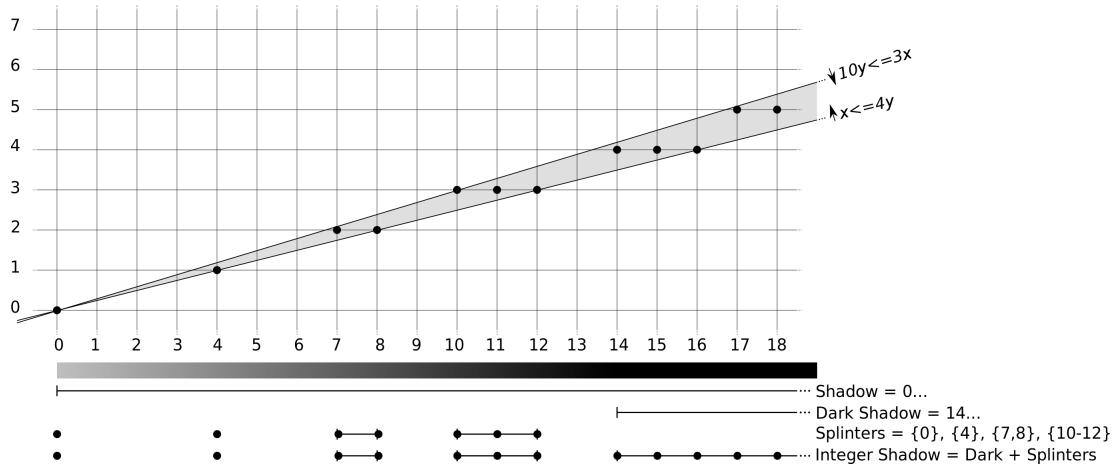


Figure 6. Shadow and Dark Shadow Metaphors for Integer Variable Elimination

$10y \leqslant 3x \wedge x \leqslant 4y$, which have integer solutions for $x = 0, 4, 7, 8, 10, 11, 12$ and $x \geqslant 14$. If Omega were to eliminate y during satisfiability testing of $10y \leqslant 3x \wedge x \leqslant 4y \wedge x \leqslant 100$, it would find that the dark shadow $(14 \leqslant x \leqslant 100)$ is satisfiable, and report True; if it were to eliminate y during satisfiability testing of $10y \leqslant 3x \wedge x \leqslant 4y \wedge x \leqslant 12$, it would find the rational shadow is satisfiable but the dark shadow isn't, and proceed to explore the splinters.

The number of inequations can grow exponentially in the number of variables eliminated by Fourier's method, and the need to compute splinters can only make matters worse. To control this problem in practice, Omega first eliminates variables that do not introduce splintering (projecting away x instead of y in Figure 6, since the rational and dark shadows on the y axis are identical), and from these non-splintering variables selects those that minimize the growth in the number of inequations. These steps appear to control the growth of inequalities during dependence analysis [Pug91a, PW94].

3.2 The Gist Operation

As noted in Section 2.1, we may wish to classify the flow dependence of Figure 1b as "conditional", since there are values of n and k for which no dependence exists and the iterations of the loop can be run in parallel. A simple definition of conditional dependence could be implemented by eliminating the loop index variables and identifying any non-tautology as conditional. For this example, $(\exists i, i' : i' = i + k \wedge 0 \leqslant i < n \wedge 0 \leqslant i' < n \wedge i < i')$, or simply $(0 < k < n)$, is not a tautology, and this dependence would thus be marked as conditional.

Unfortunately, this simple definition would categorize as "conditional" almost any dependence inside a loop with a symbolic upper bound, such as that in 1a — projecting away i and i' leaves $(2 \leqslant n)$, but of course when $n < 2$ there is no point in parallelizing the loop. To avoid such "false positives", the Omega Test defines as conditional any dependence that does not exist under some conditions when we still wish to optimize the code, e.g., when a loop we wish to parallelize runs multiple iterations.

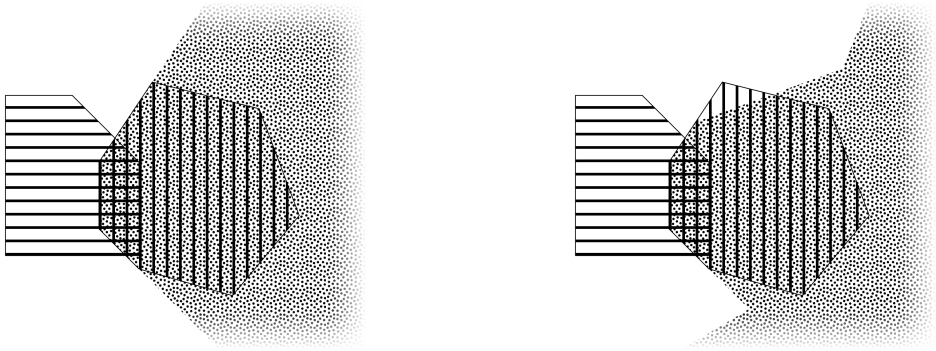
This definition makes use of the Omega Library's "gist" operation as well as projection. Informally, the "gist" of one set of constraints p (i.e., those in which a dependence exists) given some other conditions q (i.e., those in which we care about the dependence) is the "interesting" information about p given that q is true. To give precise semantics while allowing flexibility in the processing of "interesting information", gist is formally defined in terms of the values in the sets/relations being constrained, not the collection of constraints giving the bounds: "(gist p given q)" is any set/relation such that $p \wedge q = (\text{gist } p \text{ given } q) \wedge q$. Figure 7 gives a graphical illustration of the gist operation. Both examples illustrate, with a speckled region, the gist of a vertically-striped region given a horizontally-striped region. Both cases illustrate the formal definition of gist, as the speckled region overlaps the horizontally-striped region in the same way the vertically-striped region does. The option on the left illustrates the "usual" behavior that is produced by the algorithm described below: the gist is a superset of the original vertically-striped region, with simple extensions of boundaries that defined that region. The possibility of results like that on the right allows flexibility in the definition of "simple" that is important when working with non-convex sets.

When p and q are conjunctions of affine (in)equations, the Omega Library computes (gist p given q) by first using the redundancy detection steps described above to check the (in)equations of p for redundancy with respect to $p \wedge q$ (to check an (in)equation p_i, it uses p_i's hash key to determine if any other single (in)equation might make p_i redundant; checks for variables that are not bounded in one direction that thus might show p_i cannot be redundant; and (if p_i is an inequation) compares p_i with pairs of inequations). If these "quick tests" fail to classify an (in)equation as redundant or not, Omega optionally performs the definitive but potentially expensive satisfiability test of a conjunction of all (in)equations from $p \wedge q$ but with p_i negated. The conjunction of (in)equations of p that are not marked an redundant in $p \wedge q$ is then returned as $(\text{gist } p \text{ given } q)$.

Returning to the codes of Figure 1, the flow dependence from Example 1b is marked as conditional, as $(\text{gist } (0 < k < n) \text{ given } n > 1) = (0 < k < n)$. Example 1a's dependence is not, since, $(\text{gist } (2 \leqslant n) \text{ given } n > 1) = \text{True}$. More details on the gist operation, such as the algorithms used when p and q are not both conjunctions of (in)equations, can be found in [PW92, Won95, PW98].

3.3 Disjunctive Normal Form

Value-based dependence analysis, and certain cases of memory-based analysis (e.g., for programs with conditional statements) may produce constraints that are not simple conjunctions of equations and inequations.



Simple constraints for gist Complex yet legal constraints for gist

Figure 7. Examples of the Gist Operation: Usual and Unusual-but-legal Options

In Figure 3, the memory-based dependence from the write to A[i] to the read of A[i-1] is $\{[1, t, 2, i, 1] \rightarrow [1, t', 1, i', 1] \,|\, i' - 1 = i \wedge 0 \leqslant t, t' < T \wedge 1 \leqslant i, i' < N - 1 \wedge t < t'\}$, or more concisely $\{[1, t, 2, i, 1] \rightarrow [1, t', 1, i + 1, 1] \,|\, 0 \leqslant t < t' < T \wedge 1 \leqslant i \leqslant N - 3\}$. To describe the actual flow of values, the Omega Test must rule out all cases in which the value written to A[i] is overwritten before the read, i.e., any iteration $\{[1, t'', 2, i'', 1]\}$ that writes to the same location (so $i'' = i$) and occurs between the original write and the read (so $t < t'' < t'$) during a valid execution of the statement ($0 \leqslant t'' < T \wedge 1 \leqslant i'' < N - 1$); in other words it must represent the dependence relation

$$\{[1, t, 2, i, 1] \rightarrow [1, t', 1, i + 1, 1] \,|\, \begin{array}{l} 0 \leqslant t < t' < T \wedge 1 \leqslant i \leqslant N - 3 \wedge \\ \neg(\exists\, t'', i'' : i'' = i \wedge t < t'' < t' \wedge 0 \leqslant t'' < T \wedge 1 \leqslant i'' < N - 1)\} \end{array}$$

Omega handles such formulae by converting them into disjunctive normal form, i.e. a list of conjunctions of the form given in 3.1 whose disjunction is equivalent to the original formula. Conversion to disjunctive normal form can increase the size of the formula exponentially; in practice, the explosion in formula size during dependence analysis is largely due to constraints that, once negated, contradict the positive constraints (on the first line of our example formula). This problem can be controlled by applying the rule $a \wedge \neg b = a \wedge \neg(\text{gist } b \text{ given } a)$. (Refer back to Figure 7 for the intuition behind this rule — since the speckled region (gist b given a) must intersect a in exactly the way b did, so must the complement of the speckled region $\neg$(gist b given a) intersect a as $\neg b$ does.) This use of gist turns the formula above into

$$\{[1, t, 2, i, 1] \rightarrow [1, t', 1, i + 1, 1] \,|\, \begin{array}{l} 0 \leqslant t < t' < T \wedge 1 \leqslant i \leqslant N - 3 \wedge \\ \neg(t \leqslant t' - 2)\}. \end{array}$$

The inequation $t \leqslant t' - 2$ can then be negated to produce $t' \leqslant t + 1$ rather than the disjunction of eight inequations that would have arisen without gist. In this context, Omega applies only the "quick tests" for gist, as there is no point in the optional full computation here. Techniques from Section 3.1 show our example formula is satisfiable and further simplify it to our final description of the value-based flow dependence:

$$\{[1, t, 2, i, 1] \rightarrow [1, t + 1, 1, i + 1, 1] \,|\, 0 \leqslant t < T - 2 \wedge 1 \leqslant i \leqslant N - 3\}.$$

The Omega Library can apply this conversion to disjunctive normal form recursively, allowing it to operate on the full language of "Presburger Arithmetic": arbitrary formulae involving conjunction, disjunction, and negation of linear equations and inequations. Note that this logic has super-exponential complexity [FR74], and queries lacking the properties discussed above may well exhaust all available time or memory or produce integer overflow.

The Omega Test actually performs value-based analysis in a slightly different way from that described here, to "factor out" the effects of an overwrite on many different dependences, but the need for, and implementation of, negated constraints follows the principles stated here. (See [Won95, PW98] for this and other details, and [SW01] for more discussion of asymptotic complexity).

3.4 Non-Affine Terms and Uninterpreted Function Symbols

The constraint-manipulation techniques of the previous paragraphs can be used for memory-based, value-based, and conditional dependence analysis as long as the terms that are gathered from the program are affine functions of the loop bounds and symbolic constants. When some program term is not affine, many dependence tests simply report a maximally conservative approximation ("some dependence of some sort might exist here") without further analysis. The Omega Test can use two abilities of the Omega Library to produce more precise results, in many cases ruling out dependences despite the presence of non-affine terms.

The simplest approach to handling non-affine terms is to approximate the troublesome constraint but retain precise information about other constraints. The Omega Library provides a special constraint *UNKNOWN*, to be used in place of any constraint that cannot be represented exactly. Its presence indicates an approximation of some sort, but does not prevent Omega from manipulating other constraints or producing an exact final result in some cases (note that $(UNKNOWN \vee \text{True} = \text{True})$ and $(UNKNOWN \wedge \text{False} = \text{False})$; however, since the two unrepresentable constraints may arise from unrelated terms in a program, $\neg UNKNOWN$ does not contradict $UNKNOWN$, and $(UNKNOWN \wedge \neg UNKNOWN = UNKNOWN)$).

For example, suppose Figure 1b updated A[i*k] rather than A[i+k] — the constraint $i' = i + k$ must be replaced with $i' = i \cdot k$, but of course this is outside the domain of the Omega Library. (Polynomial terms in dependence constraints were explored in [MP94] and by a number of authors not related to the Omega Project, but never released in Omega.) Thus, the Omega Test instead produces the relation $\{[i] \rightarrow [i'] \mid UNKNOWN \wedge 0 \leqslant i < n \wedge 0 \leqslant i' < n \wedge i < i'\}$.

A more detailed description can be created by replacing the unrepresentable *term* rather than the entire unrepresentable *constraint*. Omega records such a term as an *uninterpreted function symbol* — a term that indicates a value that may depend on some parameters. For example, $i \cdot k$ depends only on i and k, and can thus be represented as $f(k,\ i)$ for some function f. While *UNKNOWN*s cannot be combined in any informative way, two occurrences of the same function *can* be combined in any context in which their parameters can be proved to be equal, e.g. $(f(k,\ i) > x \wedge i = i' \wedge f(k', i') < x - 5 \wedge k' = k)$ can be simplified to False. This principle holds even when f is not a familiar mathematical function, most notably *any* expression e nested in loops i_1, i_2, i_3, ... i_n can be represented $f_e(i_1, i_2, i_3, ...i_n)$. The use of function symbols can improve the precision of dependence analysis, and when this does not eliminate a dependence, the Omega Test may be able to disprove it by including user assertions involving function symbols (if the programmer has provided them).

Presburger Arithmetic with uninterpreted function symbols is undecidable in general, and the Omega Library currently restricts function parameters to be a prefix of the input or output tuple of the relation. The Omega Test uses these tuples to represent the values of loop index variables source and sink of a dependence, and thus must approximate in cases in which a function is applied to some other values, in particular the values of loop indices at the time of an overwrite that kills a value-based dependence (i.e., t'' and i'' in the relation in Section 3.3). Further detail about the treatment of non-affine constraints in the Omega Test, including empirical studies, can be found in [PW98].

4 Current Status and Future Directions

The Omega Project's constraint manipulation algorithms are described in the aforecited references and the dissertations of Bill Pugh's students. Algorithms with relatively stable implementations were generally released as the Omega Library, a freely available code base which can still be found on the Internet (an open-source version containing updates from a number of Omega Library users can be found at `http://github.com/davewathaverford/the-omega-project/`). Notable omissions from the Omega Library code base include the code for iteration-space slicing, implementations of algorithms for polynomial constraints, some "corner cases" of the full Presburger satisfiability-testing algorithm, and any way of handling cases in which the core implementation of integer variable elimination produces extremely large coefficients. The Omega Library is generally distributed with the Omega Calculator, a text-based interface that allows convenient access to the operations of the library.

The constraint-based/polyhedral approach that was pioneered by Bill Pugh's Omega Project and by Paul Feautrier and his colleagues in France remains central to a number of ongoing compiler research projects. It is also a valuable tool in industrial compilers such as that produced by Reservoir Labs, Inc.. For a discussion of the state-of-the-art before this work, see Michael Wolfe's "High-Performance Compilers for Parallel Computing" [Wol96]; additional discussion of the instance-wise approach to reasoning about program transformations can be found in Jean-Francois Collard's "Reasoning About Program Transformations: Imperative Programming and Flow of Data" [Col02].

A number of constraint-manipulation program analysis/transformation techniques and associated libraries have been developed since the release of the Omega Library. These typically contain more modern algorithms for a number of advanced functions of Omega, such as code generation, or more general implementations of underlying constraint algorithms, for example avoiding Omega's use of limited-range integers. As of the writing of this article, most do not support all of the techniques described in this article, notably

- algorithms that are (at least in principle) exact for integer variables;
- algorithms for the full domain of Presburger Arithmetic;
- separation of dependence analysis and transformation from the core constraint system via
  - an API allowing high-level relation/set operations and low-level operations and
  - a text-based interface to allow easy exploration of the abilities of the system;
- special terms for communicating with the core constraint system about information not in the its primary domain, e.g. function symbols;
- tagging of approximations to distinguish them from exact results; and
- iteration space slicing (never released with Omega).

Bibliography

[Col02] Jean-Francois Collard. *Reasoning about Program Transformations*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[Fea88] Paul Feautrier. Array expansion. In *ICS*, pages 429–441, 1988.

[FR74] Michael J. Fischer and Michael O. Rabin. Super-exponential complexity of Presburger arithmetic. In Richard M. Karp, editor, *Proceedings of the SIAM-AMS Symposium in Applied Mathematics*, volume 7, pages 27–41, Providence, RI, 1974. AMS.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[KP94] Wayne Kelly and William Pugh. Finding legal reordering transformations using mappings. In *LCPC*, pages 107–124, 1994.

[KP96] Wayne Kelly and William Pugh. Minimizing communication while preserving parallelism. In *International Conference on Supercomputing*, pages 52–60, 1996.

[KPRS95] Wayne Kelly, William Pugh, Evan Rosser, and Tatiana Shpeisman. Transitive closure of infinite graphs and its applications. In *LCPC*, pages 126–140, 1995.

[MP94] Vadim Maslov and William Pugh. Simplifying polynominal constraints over integers to make dependence analysis more precise. In *CONPAR*, pages 737–748, 1994.

[PR97] William Pugh and Evan Rosser. Iteration space slicing and its application to communication optimization. In *International Conference on Supercomputing*, pages 221–228, 1997.

[PR99] William Pugh and Evan Rosser. Iteration space slicing for locality. In *LCPC*, pages 164–184, 1999.

[Pug91a] William Pugh. The Omega Test: a fast and practical integer programming algorithm for dependence analysis. In *SC*, pages 4–13, 1991.

[Pug91b] William Pugh. Uniform techniques for loop optimization. In *ICS*, pages 341–352, 1991.

[Pug92] William Pugh. A practical algorithm for exact array dependence analysis. *Commun. ACM*, 35(8):102–114, 1992.

[PW92] William Pugh and David Wonnacott. Eliminating false data dependences using the Omega test. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 140–151, San Francisco, California, June 1992.

[PW94] William Pugh and David Wonnacott. Experiences with constraint-based array dependence analysis. In *Principles and Practice of Constraint Programming, Second International Workshop*, volume 874 of *Lecture Notes in Computer Science*, pages 312–325. Springer-Verlag, Berlin, May 1994. Also available as Tech. Report CS-TR-3371, Dept. of Computer Science, University of Maryland, College Park.

[PW98] William Pugh and David Wonnacott. Constraint-based array dependence analysis. *ACM Trans. Program. Lang. Syst.*, 20(3):635–678, 1998.

[SW01] Robert Seater and David Wonnacott. Polynomial time array dataflow analysis. In *LCPC*, pages 411–426, 2001.

[Wol96] Michael Wolfe. *High Performance compilers for parallel computing*. Addison-Wesley publishing Co., 1996.

[Won95] David G. Wonnacott. *Constraint-Based Array Dependence Analysis*. PhD thesis, Dept. of Computer Science, The University of Maryland, August 1995. Available as refftp://ftp.cs.umd.edu/pub/omega/davewThesis/davewThesis.ps.

[Won00] David Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *IPDPS*, pages 171–180, 2000.

[Won02] David Wonnacott. Achieving scalable locality with time skewing. *International Journal of Parallel Programming*, 30(3):1–221, 2002.