Exact Data Dependence Analysis Using Data Access Descriptors

Lorenz Huelsbergen
lorenz@cs.wisc.edu

Douglas Hahn
hahn@tekcrl.labs.tek.com

James Larus
larus@cs.wisc.edu

Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton St.
Madison, Wisconsin 53706

December 6, 1993

Abstract

Data Access Descriptors provide a method for summarizing and representing the portion of an array accessed by a program statement. A Data Access Descriptor does not, however, indicate if its characterization is exact or conservative, nor does it record the temporal order of accesses. Exactness is necessary to expose maximal parallelism. Temporal information is necessary to calculate *direction vectors* for inter-loop dependences.

This paper presents an extension to basic Data Access Descriptors that identifies exact representations. We illustrate the value of extended Data Access Descriptors by showing how to calculate information typically provided by direction vectors and by refining potential conflicts between statements with *array kill* information.

Keywords: data dependence analysis, Data Access Descriptors, direction vectors, loop analysis, compilers, parallel computers.

1 Introduction

Compiler optimization of array-manipulating programs and parallelization of program loops depends on accurately detecting and precisely categorizing data dependences between array references. The efficiency of this analysis depends on the underlying representation of the array elements referenced by a statement. Some representations allow efficient detection of dependences at the expense of efficient summarization. Since large programs generate an unwieldy number of data dependences, summarization is essential for efficient dependence analysis. On the other hand, compromising precision for efficiency raises the possibility of overlooking potential optimization and parallelization opportunities. The descriptor used in this paper allows efficient summarization and calculation of data dependences and other types of array analysis information.

Our work extends Balasundaram's Data Access Descriptors (DADs) [Bal89a, Bal90] with additional operations that compute dependence analysis information previously unavailable in this representation. The modified representation allows efficient summarization as well as efficient calculation of data dependences and other types of array analysis information. This includes calculation of the *direction* of dependences, *array kill* information, and a temporal ordering of accesses. To support these extensions, we provide an efficient test that characterizes whether summarized accesses are *exactly* or conservatively represented by a DAD.

1.1 Data Access Descriptors (DADs)

Balasundaram's DADs are derived from the work of Callahan and Kennedy on *Regular Sections* [Cal86]. While Callahan's Regular Section Descriptors are very general—they include information on bounds of iteration variables and original subscript expressions—they are inefficient. The primary advantage of DADs is efficiency. Intersection of Data Access Descriptors computes data conflicts and union of Data Access Descriptors approximates summary information. Both operations require constant time if the number of array dimensions is bounded. Appendix A provides an overview of DADs.

The component of DADs relevant to our work is the simple section, an n-dimensional polytope describing the portion of an array referenced by an n-dimensional access. A simple section is composed of a pair of bounding planes for each array dimension. These bounding planes are efficiently computed by a variation of Banerjee's inequality [Ban88]. Figure 1 delineates

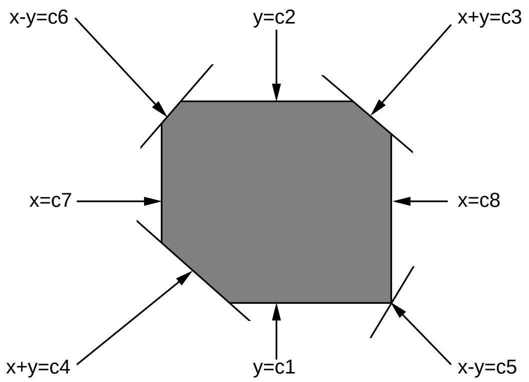

Figure 1: A simple two-dimensional section and its boundaries. The simple section is the shaded area enclosed by the eight lines, all of which are of the form $x = c$, $y = c$, or $x \pm y = c$.

the boundaries of a simple section that describes a two-dimensional array access.

DADs can efficiently summarize array accesses over several statements or entire subprograms. The ability to summarize the effects of multiple statements, together with DADs' ability to test for data dependences with simple, linear algorithms, make DADs a good candidate for a general representation of array references. However, DADs cannot determine some information easily computed by conventional data-dependence analysis techniques. Our extensions to DADs rectify this problem by computing the *direction vector* [Wol89] for a dependence and precise *array kill* information.

1.2 Exact Information

Exact information is necessary for some compiler analyses (e.g., deciding if one loop kills the definitions produced by another) and is generally desirable in applications of dependence information. Furthermore, inexact information may prevent the detection of *any* parallel execution orders since a false dependence can cause independent statements to appear to conflict. Our work detects when summary information becomes inexact, and hence allows the use of optimizations that require precise information.

A summary is *exact* if it describes all referenced array elements and includes no extraneous, unreferenced elements. Combining summarized information from multiple DADs with Balasundaram's union operator sometimes produces an inexact, conservative representation of array references. Conservative approximation is unavoidable in general, however it is useful to know when a representation is exact and when it is conservative.

Since simple sections are not closed under set union, it cannot be known whether the simple section approximating the set union of two arbitrary simple sections is exact. The union of inexact simple sections can produce an exact simple section. However, determining if the union of two *exact* simple sections can be precisely represented by a simple section is possible and an algorithm is described below.

For example, Figure 2 depicts simple sections representing rectangular and triangular access in a two dimensional array. The simple section S precisely describes an access of the form:

```
for i ← 0 to N do
    for j ← 0 to N do
        ...A[i, j] ...
    endfor
endfor
```

S' precisely represents an access similar to:

```
for i ← L to M do
    for j ← 0 to i - L do
        ...A[i, j] ...
    endfor
endfor
```

It may be convenient to summarize both accesses as a single simple section—for example if both loops occur in a subprogram. If S and S' in Figure 2a are both exact, the simple section summarizing their effects is exact (and is equivalent to S). However, Figure 2b depicts slightly different simple sections S and S' whose summary simple section is inexact since it contains elements neither loop accesses (those lying in the concavity produced by S and S'). Unextended DADs conservatively characterize both summary sections as inexact. Extended DADs detect the exactness of the section in Figure 2a, potentially exposing potent optimizations.

A new polynomial-time algorithm, **union_exact**, determines if a DAD is an exact summary of several exact accesses. This algorithm proceeds by comparing corresponding bounds of the regions described by two operand DADs. As the union DAD is constructed, certain constraints are imposed

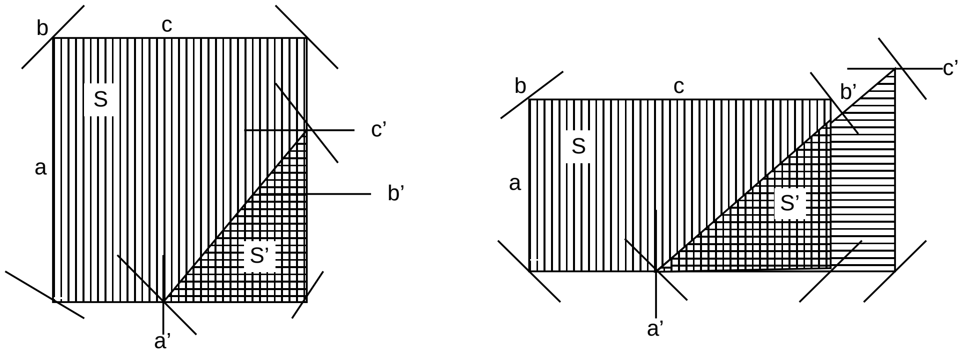

Figure 2: Examples of (a) exact and (b) inexact simple section unions.

by **union_exact** on the remaining unprocessed boundaries, i.e., a boundary must be selected for the union DAD, or may not be selected for the union DAD. If these constraints are violated, the resulting union is inexact. The algorithm is amenable to integration with the original DAD union operation.

1.3 New DAD Operations

A direction vector indicates whether a dependence exists in the same iteration of a loop, crosses iteration boundaries forwards or backwards, or is a combination of these. Array kill information describes the portions of an array defined by an assignment statement. Over the course of a loop, an array assignment may overwrite the values written by previous assignments into the array. Most conventional data-flow analysis techniques (e.g., def/use chains) produce overly conservative array kill information since exact array accesses are not identified or categorized as such. Our extensions efficiently provide accurate direction vectors and array kill descriptions. These extensions are based on *temporal slices*, which are computed by a simple variation of Balasundaram's algorithm for computing simple sections.

Temporal slices are simple sections with symbolic bounds that describe the elements of an array accessed prior to, during, and after an arbitrary loop iteration. This ordering permits calculation of the direction of a dependence. For example, if the slice representing the locations accessed by a statement in the *present* loop iteration conflicts (e.g., intersects) with the *past* slice of a write statement, the read statement is data *flow-dependent* on the write statement.

Temporal ordering information is also used in an algorithm that computes array kill information for exact write references. Examples show that this analysis is better than conventional techniques since a DAD can represent the killed array elements more precisely.

1.4 Overview

This paper is divided into two parts. First, we describe extensions that allow detection of exactness when performing DAD union (Section 2). We then present dataflow analysis techniques that use extended simple sections and exactness knowledge to compute dependence direction information (Section 3) and array kill information (Section 4). Appendix A reviews Balasundaram's work regarding DADs. Appendix B proves the correctness of our exact union test.

2 Exact Union Test

Since simple sections are not closed under set union (see Appendix A), it cannot be known whether the simple section approximating the set union of two simple sections is exact. The union of inexact simple sections may produce an exact simple section. Detecting this is impossible due to information lost by the DAD abstraction. Determining whether the union of two exact simple sections can be precisely represented by a simple section involves restricting possible boundary choices for the resulting simple section. This section develops a polynomial-time algorithm ($O(n^2)$) in the number of array dimensions, n, that determines if these conditions hold.[1] This algorithm is amenable to integration with the standard simple section union algorithm.

2.1 Exactness Criteria

The algorithm for simple section union selects appropriate boundaries for the resulting simple section from the simple sections supplied as operands. If the result is to be exact, the selection of certain boundaries may require the selection of other boundaries from the same operand. If this is the case, the first boundary b is said to *constrain* the other boundary b'.

Before any simple section boundaries are selected for the union, all boundaries are unconstrained and may be selected from either constituent

[1] In most programs, n is limited to 3 or 4, so the algorithm runs in constant time

simple section. As a boundary is selected from one of the simple sections, "neighboring" boundaries from the same simple section become constrained. These neighboring boundaries are termed *adjacent*.

Informally, a boundary b is adjacent to a boundary b' if b and b' are not parallel or perpendicular to one another and b and b' both contain a common point in the simple section they define. Referring to Figure 1, boundaries $x - y = c_6$ and $x + y = c_3$ are adjacent to the boundary $y = c_2$. Note that $x = c_8$ and $x = c_7$ are not adjacent to $y = c_2$ since they are perpendicular. Similarly, boundaries $x = c_7$ and $y = c_2$ are adjacent to $x - y = c_6$.

The key observation is that if a boundary b is chosen for the simple section union, b's adjacent boundaries must be included as well. Figure 2 provides an example. Figure 2a shows two simple sections of dimension $n = 2$, S and S', that form an exact union. Boundary b is chosen over b' since it is the "more exterior" ($b < b'$—it is a lower boundary). This choice imposes the constraints that boundaries a and c be chosen as well to maintain an exact union. In Figure 2b, boundary b again constrains the choice between a and a' to a, and between c and c' to c. However, the constraint is violated because the union algorithm selects c', which is more exterior, over c.

A special case arises when the two corresponding boundaries, b and b', are of identical magnitude. Neither is "more exterior" to the other. Here, we must verify that the corresponding simple sections induced by b and b' form exact simple sections in one-dimensional space (or two-dimensional space if the simple sections are of dimension $n > 2$). This can be viewed as successively checking all the lower dimensional simple sections that have b and b' as boundaries. Immediately reducing the check to one- or two-dimensional space is valid because simple section boundaries are defined in terms of no more than two coordinate axes. This check insures that no "gaps" exist between the two simple sections, or, in other words, that an overlap exists between the sections in the hyperplanes induced by the equal bounds. If all such lower dimensional simple sections are exact, the resulting simple section union is *sub-section consistent*.

A simple section S is *boundary consistent* with respect to its constituent simple sections if the adjacency and sub-section conditions are not violated.

> **Theorem 1** The union of two simple sections, $S = S_1 \cup_{simp} S_2$, is *exact* if and only if S is boundary consistent.

A proof is presented in Appendix B.

```
boolean function union_exact (simplesection S_1, S_2, integer n);
  simplesection T_1, T_2;

  if not exact(S_1) or not exact(S_2) then
    return false;
  if n = 1 then
    return overlap(S_1, S_2);
  unmark (S_1); unmark (S_2);
  for each simple boundary pair B_i = α_i ≤ ψ(x_1, ..., x_n) ≤ β_1 ∈ S_1 do
    pick its corresponding boundary pair B'_i = α'_i ≤ ψ(x_1, ..., x_n) ≤ β'_i ∈ S_2;
    if β_i < β'_i then
      if marked(β_i) then return false;
      else
        begin
          mark(β'_i);
          markadjacent(β'_i);
        end
    else if β'_i < β_i then
      if marked(β'_i) then return false;
      else
        begin
          mark(β_i);
          markadjacent(β_i);
        end
    else
      begin
        for all simple sections in min(n − 1, 2) space do
          T_1 ← simple section induced by β_i
          T_2 ← corresponding simple section induced by β'_i
          if not union_exact(T_1, T_2, min(n − 1, 2)) then
            return false;
        endfor
      end
  endfor
  return true;
end union_exact;
```

Figure 3: Algorithm **union_exact**.

2.2 Exact Test Algorithm

Theorem 2.1 provides the basis for an algorithm to determine whether the simple section union of two exact simple sections is exact. This algorithm is presented in Figure 3.

Parameters to the function **union_exact** are two simple sections, S_1 and S_2, and their dimension, n. If $S_1 \cup_{simp} S_2 = S_1 \cup S_2$ the function returns the Boolean value *true*; otherwise it returns *false*. Initially S_1 and S_2 are examined to ensure that they are exact. If either simple section is inexact, exactness of the resulting union cannot, in general, be determined. In this case, the algorithm returns the conservative answer. If the simple sections are one-dimensional, an auxiliary function **overlap** indicates whether S_1 and S_2 abut or contain a common point. This test suffices for one-dimensional exactness.

The algorithm continues by **unmark**ing the simple boundaries of both simple sections. A mark indicates that a constraint has arisen requiring subsequent selection of the marked boundary. Initially all boundaries are unconstrained, hence unmarked.

Next, the algorithm compares corresponding boundary pairs. If a boundary is selected as participating in the boundary of the resulting union, a check is made to ensure that the boundary not marked is not required to participate in the union (by a boundary encountered earlier). When a boundary is selected, it and its adjacent boundaries are constrained by **mark** and **markadjacent** respectively.

In case the corresponding boundaries being examined are equivalent, all corresponding simple sections in lower dimensions induced by the equal boundaries are compared via recursive calls to **union_exact**.

Algorithm **union_exact** requires extensions to simple sections allowing a flag indicating whether the simple section is known to be exact and a flag marking every simple boundary.

The algorithm is quadratic in the number of dimensions n. Each call to **union_exact** may iterate through all $2n^2$ boundary pairs performing a worst-case bounded number of calculations for each. If the number of dimensions is fixed, as suggested in [Bal89a], time and space requirements are constant.

3 Dependence Analysis

DADs summarize all array references made by a statement or set of statements during the execution of a loop. The presence of a *data dependence* can be detected by intersecting two simple sections. By contrast, conventional methods such as Banerjee's Exact Test [Ban76] compare the statements' array index expressions and attempt to find a common integer solution. While intersection of DADs detects dependences, more detail is necessary for most uses of dependence information. Knowledge of a dependence's type, its direction, and the loops that carry it permit aggressive optimization.

Three types of dependences connect program statements. A statement S_r that reads a memory location previously written by statement S_w is *flow-dependent* on S_w. A memory location read in statement S_r and subsequently reassigned in S_w makes S_w *anti-dependent* on S_r. Finally, S_w is *output-dependent* on S_v if S_w and S_v write to the same memory location and S_v precedes S_w.

Further detail is provided by *direction vectors*, which indicate whether a dependence exists in the same iteration of a loop, crosses iteration boundaries forwards or backwards, or is comprised of a combination of these. Conventional analysis computes a direction vector with the Banerjee-Wolfe Algorithm [Wol89]. This algorithm uses Banerjee's inequality to discover possible dependences by incrementally refining induction variable value-ordering between linearized subscript equations. For example, if two references are enclosed in a doubly-nested loop, the algorithm first checks for dependences with no constraints. If a dependence is discovered, the outer loop's induction variable in the first statement is constrained to be greater than, equal to, or less than $((>, *), (=, *), (<, *))$ the variable in the second statement. For each dependence thus discovered, the algorithm constrains the inner loop's induction variable. When complete, this test establishes the conditions under which a dependence exists. For example, it may occur when the outer loop induction variable is less ('$<$') and the inner is greater ('$>$') in the first reference than in the second reference expression. If the first reference writes and the second reads, the dependence is a flow-dependence and the outer loop carries it.

Information this detailed cannot be gathered directly from DADs because simple sections are computed by the Banerjee inequality without constraints. While this test is potentially more accurate since subscripts are not linearized, it is equivalent to the unconstrained, top-level Banerjee-Wolf test. Since simple sections also abstract references to the induction variable,

```
function temporal_slice ();
   for j ← 1 to k do
      past_j(p)    ← ⟨L_j ≤ x_j ≤ I_j − 1⟩;
      present_j(p) ← ⟨I_j ≤ x_j ≤ I_j⟩;
      future_j(p)  ← ⟨I_j + 1 ≤ x_j ≤ U_j⟩;
   endfor
end temporal_slice;
```

Figure 4: Algorithm **temporal_slice** computes the primitive temporal slices for a statement containing an n dimensional array access, $\mathrm{A}[d_1, \ldots, d_n]$, that is enclosed in k loops with lower bounds $(L_1, \ldots, L_k)$, upper bounds $(U_1, \ldots, U_k)$, and induction variables $(I_1, \ldots, I_k)$.

constraints cannot later be imposed on the geometric intersection operation. Central to the computation of dependence information from DADs is the inclusion of temporal information.

3.1 Temporal Slices

Simple sections must be extended to include temporal information. This information can be represented by triples of simple sections. Each component section is a *temporal slice* of an array reference. Temporal slices with respect to a single loop are *primitive*. Slices of multiple nested loops are *composite*.

The algorithm in Figure 4 constructs the primitive slices: $\mathbf{past}_j(S)$, $\mathbf{present}_j(S)$, and $\mathbf{future}_j(S)$ with respect to loop j, $1 \leq j \leq k$, for a statement S containing an n dimensional array access, $\mathrm{A}[d_1, \ldots, d_n]$, enclosed in k loops with lower bounds $(L_1, \ldots, L_k)$, upper bounds $(U_1, \ldots, U_k)$, and induction variables $(I_1, \ldots, I_k)$.

The three temporal slices represent the portion of an array accessed in the loop iterations prior to an arbitrary iteration, during that iteration, and in subsequent iterations. $\mathbf{past}_j$ is a simple section containing elements of A accessed in preceding iterations of the current invocation of loop j (ignoring surrounding loops). Elements accessed in the current iteration of this loop are described by the $\mathbf{present}_j$ slice. The $\mathbf{future}_j$ slice contains elements referenced in subsequent iterations of the current invocation of loop j. For each loop j in a k-loop nest, the algorithm substitutes appropriate loop bounds for the jth induction variable, I_j, in the symbolic simple section

$$
\begin{aligned}
T_j, \mathbf{past}_{j+1} &= \left(T_j \cap \bigcup_{1 \le i \le j} (\mathbf{past}_i \cup \mathbf{present}_i) \right) \cap \mathbf{past}_{j+1} \\
T_j, \mathbf{present}_{j+1} &= T_j \cap \mathbf{present}_{j+1} \\
T_j, \mathbf{future}_{j+1} &= \left(T_j \cap \bigcup_{1 \le i \le j} (\mathbf{present}_i \cup \mathbf{future}_i) \right) \cap \mathbf{future}_{j+1}
\end{aligned}
$$

Figure 5: Rules for constructing composite temporal slices from primitive slices. T_j is a temporal slice whose last component is computed with respect to loop j.

$\langle d_i \le x_i \le d_i \rangle$.

past, **present**, and **future** primitive temporal slices can be combined into composite slices by the composition rules of Figure 5. Composite slices are formed from primitive slices by moving from the outermost to innermost enclosing loop. If a temporal slice, T, describes elements accessed by the first j loops, intersection with $\mathbf{present}_{j+1}$ produces a slice defining the elements referenced in the present iteration of loop $j+1$ (rule 2). Adjoining other slices from loop $j+1$ constrains the elements in slices from surrounding loops. For example, adding $\mathbf{past}_{j+1}$ eliminates elements in $\mathbf{future}_i$ for $1 \le i \le j$ since these elements cannot be referenced prior to the current invocation of the $j+1$ loop.

For a pair of nested loops i and j, the meaningful composite slices have the following interpretations:

	$\mathbf{past}_j$	$\mathbf{present}_j$	$\mathbf{future}_j$
$\mathbf{past}_i$	Before current invoc. of j loop		
$\mathbf{present}_i$	Earlier in current invoc. of j loop	During current execution of statement	Later in current invoc. of j loop
$\mathbf{future}_i$			Subsequent invoc. of j loop

For example, applying the temporal slice algorithm to the loop:

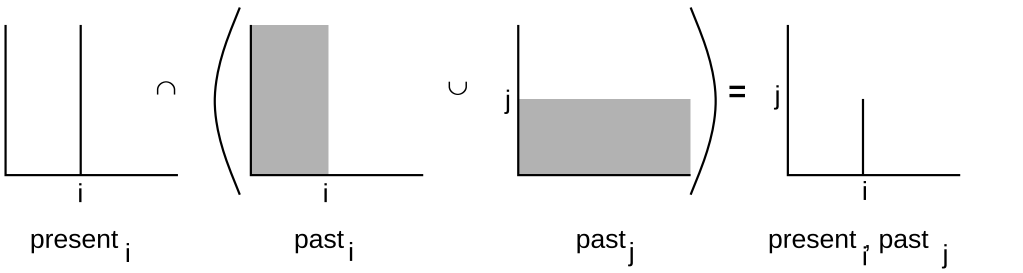

Figure 6: Primitive slices $\mathbf{present}_i$, $\mathbf{past}_i$, and $\mathbf{past}_j$ used in forming the composite temporal slice $\mathbf{present}_i, \mathbf{past}_j$.

```
for i ← 0 to N do
    for j ← 0 to M do
        A[i, j]  ← ...                            /* 1 */
                 ← ... A[i+1, j+1] ...            /* 2 */
    endfor
endfor
```

generates temporal slices for the **past**, **present**, and **future** of the array accesses in statements 1 and 2. For each access, the algorithm generates two sets of slices—with respect to the i and j loops. Some of the slices are:

	$\mathbf{past}_i$	$\mathbf{future}_i$	$\mathbf{present}_i, \mathbf{present}_j$	$\mathbf{present}_i, \mathbf{past}_j$
(1)	$\langle 0 \le x_1 \le i-1 \rangle$ $\langle 0 \le x_2 \le M \rangle$	$\langle i+1 \le x_1 \le N \rangle$ $\langle 0 \le x_2 \le M \rangle$	$\langle i \le x_1 \le i \rangle$ $\langle j \le x_2 \le j \rangle$	$\langle i \le x_1 \le i \rangle$ $\langle 0 \le x_2 \le j-1 \rangle$
(2)	$\langle 1 \le x_1 \le i \rangle$ $\langle 1 \le x_2 \le M+1 \rangle$	$\langle i+2 \le x_1 \le N+1 \rangle$ $\langle 1 \le x_2 \le M+1 \rangle$	$\langle i+1 \le x_1 \le i+1 \rangle$ $\langle j+1 \le x_2 \le j+1 \rangle$	$\langle i+1 \le x_1 \le i+1 \rangle$ $\langle 1 \le x_2 \le j \rangle$

$\mathbf{past}_i$ and $\mathbf{future}_i$ are primitive slices. The slice $\mathbf{present}_i, \mathbf{present}_j$ follows from application of rule 2. The composite slice $\mathbf{present}_i, \mathbf{past}_j$ is computed from $\mathbf{present}_i$, $\mathbf{past}_i$, and $\mathbf{past}_j$ by rule 1. Figure 6 geometrically illustrates the elements described by these primitive temporal slices and the resulting (composite) slice obtained by intersecting $\mathbf{present}_i$ with the union of $\mathbf{past}_i$ and $\mathbf{past}_j$.

3.2 Computing Data Dependences

A data dependence exists between two statements if the simple sections describing the accesses overlap, i.e. their intersection is non-empty. Table 1 categorizes the dependence according to which pairs of temporal slices of

$W \cap R \neq \emptyset$		Dependence
present(R)	**past**(W)	flow
present(R)	**present**(W)	flow/anti
present(R)	**future**(W)	anti

Table 1: A data dependence exists between an assignment (W) and a read (R) statement if the intersection of the respective DAD sections are non-empty. The *type* of dependence is detected by intersecting temporal slices.

the respective statements contain common elements. In the above example, the loop-carried anti-dependence between statements 1 and 2 is detected upon intersection of elements written by statement 1 during this invocation of the loop, $\mathbf{present}_i$,$\mathbf{present}_j(1)$, with the elements read in the "past" of statement 2, $\mathbf{past}_i$,$\mathbf{past}_j(2)$.

As a program text is lexically processed, detect and and classify data dependences by intersecting the **present** slices of previously encountered references with the **past**, **present**, and **future** slices of the current statement.

3.3 Computing Direction Vectors

Direction vectors [Wol89] are extracted from temporal slices in the following manner. Given a data dependence between two references T and S, the presence of elements in $\mathbf{past}_i(T) \cap \mathbf{present}_i(S)$ corresponds to an '<' entry for the i^{th} loop's component in the direction vector. This indicates that the dependence crosses an iteration boundary in the forward direction. Similarly, a non-empty intersection of $\mathbf{present}_i(T)$ with $\mathbf{present}_i(S)$ corresponds to an '=' in the direction vector (dependence exists in the current loop invocation) and $\mathbf{future}_i(T) \cap \mathbf{present}_i(S) \neq \emptyset$ to an '>' (dependence crosses iteration boundary backwards). The temporal relationship between statements is implicit in the slices used to compute the intersection.

3.4 Computing Temporal Slices

Temporal slices can be computed efficiently. The lifetime of a temporal slice spans only the enclosing loop, inducing no long term storage cost. If references within the loop are processed in lexical order, only **present** slices need be saved between statements.

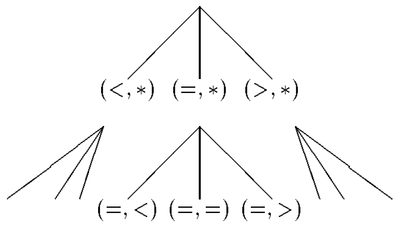

Figure 7: Banerjee-Wolf tree for two enclosing loops.

The method and time complexity of computing temporal slices is identical to that of initially generating the simple sections for the references. By judicious saving of intermediate slices, Balasundaram's simple section generation algorithm can additionally compute all "useful" temporal slices. In a doubly-nested loop, for example, temporal slices equivalent to the direction vectors listed in Figure 7 are generated at no cost during computation of the original simple section.

Since the temporal slices are defined symbolically, boundaries compared by the simple section operators may be incomparable due to insufficient constraints between symbolic bounds. If this is the case, the most conservative bounds are used, which may introduce unnecessary dependences.

4 Array Kills

An advantage of extended DADs is that they indicate whether a simple section *exactly* represents an array reference. A reference that is an assignment and is exact, kills previous assignments to a portion of an array. This means that over the course of a loop, the assignment overwrites the effects of previous assignments into a portion of the array. By computing array kills, we can refine the dependence relation by eliminating conflicts that are killed by intervening assignments. However, to compute kill information we need exact DADs since conservative descriptions produce incorrect kills.

Simple sections summarize an access over an *entire* loop's execution. Assertions about interference between statements within a loop—whether they overwrite each others effects or not—cannot be made on the basis of an overlap between simple sections. Consider the following loop:

```
for i ← 1 to N
           ← A[i]       /* 1 */
    A[i]   ←            /* 2 */
           ← A[i-1]     /* 3 */
    A[i+2] ←            /* 4 */
           ← A[i-1]     /* 5 */
endfor
```

In this loop, conventional dependence analysis and naive use of DADs would incorrectly indicate flow-dependences from statement 4 to statements 3 and 5, when in fact, statement 2 kills the definition. However, by employing temporal slices it is possible to determine how array references overwrite each other, effectively calculating array kills.

Calculating array reaching definitions using extended DADs is similar to the technique described by Gross and Steenkiste [GrSt90]. The DAD, however, is a generalization of their rectangular descriptor and is able to precisely describe a larger class of array references.

4.1 Array Reaching Definitions

The algorithm below refines an assignment statement's temporal sections with a reaching-definition data-flow analysis. Exact assignments to an array may supersede other assignment's definitions. For example, if an exact array assignment at statement T reaches an assignment S, the algorithm refines the **past**(S) slice to include only those elements not defined by T—the elements in the **past** of S not defined in the **past** or **present** of T.

Throughout this section, we assume that each statement reads or writes all elements in an array A exactly once, so for a statement S, the DAD is exact and yields the invariant : $\mathbf{past}(S) \cup \mathbf{present}(S) \cup \mathbf{future}(S) = A$. It follows that the intersection of a section X with any two of a statement's three sections is the set difference of X and the third section. For example, $X \cap (\mathbf{present}(S) \cup \mathbf{future}(S)) = X - \mathbf{past}(S)$.

The array kill algorithm computes a form of reaching definitions among array assignments through the equation: $Def'_s = (Def_s \mathbin{\dot{\cap}} Refine_s) \cup Gen_s$. The confluence operator $\dot{\cap}$ is an intersection in which tuples in the left-hand set that do not match a tuple for the same statement in the right hand set are put in the result.

Propagated definitions are four-tuples: $\langle S,\, pa_S,\, pr_S,\, f_S \rangle$ containing a statement S, its past slice pa_S, its present slice pr_S, and its future slice f_S.

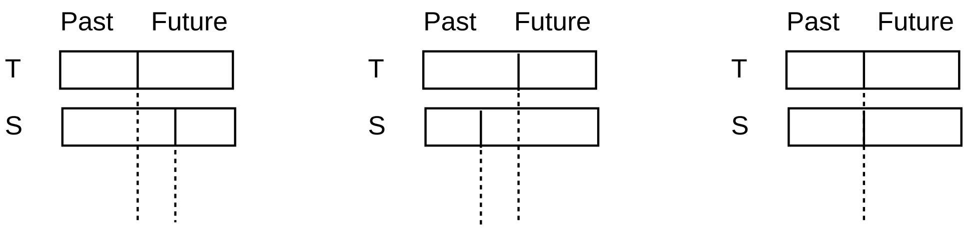

Figure 8: Three cases in calculating the *Refine* set. Statements T and S are assignments whose temporal slices overlap. In the first case, S is ahead of T. In the second case it is behind. And, in the third case, they traverse the array in lockstep.

The intersection and union of sets of these tuples is the component-wise intersection or union of tuples for the same statement.

The set Refine_S contains the portions of the array defined by a previous statement T that is redefined by the current statement S:

$$\begin{aligned} \mathit{Refine}_S &= \{\langle T,\, -,\, -,\, pa_S \cup pr_S\rangle \mid pa_S \cap pr_T \neq \emptyset \text{ and } T \text{ is exact}\} \\ &\cup \{\langle T,\, pr_S \cup f_S,\, -,\, -\rangle \mid pr_T \cap f_S \neq \emptyset \text{ and } S \text{ is exact}\} \\ &\cup \{\langle T,\, \emptyset,\, \emptyset,\, -\rangle \mid pr_T \cap pr_S \neq \emptyset \text{ and } S \text{ is exact}\} \end{aligned}$$

'$-$' is the identity element under intersection—a descriptor of the entire array A. The one-dimensional version of this problem is illustrated in Figure 8.

The set Gen_S contains the portions of the array defined by S in previous iterations that have not been killed by T:

$$\begin{aligned} \mathit{Gen}_S &= \{\langle S,\, (pr_T \cup f_T) \cap pa_S,\, pr_S,\, f_S\rangle \mid pa_S \cap pr_T \neq \emptyset \text{ and } T \text{ is exact}\} \\ &\cup \{\langle S,\, pa_S,\, pr_S,\, (pa_T \cup pr_T) \cap f_S\rangle \mid pr_T \cap f_S \neq \emptyset \text{ and } S \text{ is exact}\} \\ &\cup \{\langle S,\, pa_S,\, pr_S,\, \emptyset\rangle \mid pr_T \cap pr_S \neq \emptyset \text{ and } S \text{ is exact}\} \end{aligned}$$

Since temporal slices for a statement are relative to the current loop iteration, slices propagated along loop backedges in the dataflow framework must be "shifted." In particular, only the pa_S slice is allowed to reach a loop header along a backedge. This slice represents elements defined in the loop body that remain live throughout the current iteration of the loop, thereby reaching the header of the loop.

Another view of this problem is an algorithm to directly compute modifications to the temporal slices (Figure 9). The algorithm produces **past′**, **present′**, and **future′** slices at each assignment statement. Note that while

for array assignment statements T, S such that T reaches S along all paths **do**
  if $\mathbf{past}(S) \cap \mathbf{present}(T)$ and assignment T is exact **then**
    / assignment T kills a portion of assignment S (type I) */*
    $\mathbf{past}'(S) \leftarrow \mathbf{past}(S) \cap (\mathbf{present}(T) \cup \mathbf{future}(T))$
    $\mathbf{future}'(T) \leftarrow \mathbf{future}(T) \cap (\mathbf{past}(S) \cup \mathbf{present}(S))$

  if $\mathbf{future}(S) \cap \mathbf{present}(T)$ and assignment S is exact **then**
    / assignment S kills a portion of assignment T (type II) */*
    $\mathbf{past}'(T) \leftarrow \mathbf{past}(T) \cap (\mathbf{present}(S) \cup \mathbf{future}(S))$
    $\mathbf{future}'(S) \leftarrow \mathbf{future}(S) \cap (\mathbf{present}(T) \cup \mathbf{past}(T))$

  if $\mathbf{present}(S) \cap \mathbf{present}(T)$ and assignment S is exact **then**
    / assignment S kills assignment T (type III) */*
    $\mathbf{past}'(T) \leftarrow \emptyset$
    $\mathbf{present}'(T) \leftarrow \emptyset$
    $\mathbf{future}'(S) \leftarrow \emptyset$
endfor

Figure 9: Algorithm **array_kill**

exact sections can kill inexact sections, the opposite is not true.

After computing the array kills, the refined DADs can be used to compute better dependence information. Dependences are found by the methods of Section 3.2 and classified via the relations of Table 1.

The following loop demonstrates the operation of algorithm **array_kill**:

```
for i ← 1 to N
             ←  A[i]      /* 1 */
     A[i]    ←            /* 2 */
             ←  A[i-1]    /* 3 */
     A[i+2]  ←            /* 4 */
             ←  A[i-1]    /* 5 */
     A[i]    ←            /* 6 */
             ←  A[i+3]    /* 7 */
endfor
```

The **past**, **present**, and **future** slices input of the assignment statements are:

	past	present	future
(2)	$\langle 1 \leq x_1 \leq i-1 \rangle$	$\langle i \leq x_1 \leq i \rangle$	$\langle i+1 \leq x_1 \leq N \rangle$
(4)	$\langle 1 \leq x_1 \leq i+1 \rangle$	$\langle i+2 \leq x_1 \leq i+2 \rangle$	$\langle i+3 \leq x_1 \leq N+2 \rangle$
(6)	$\langle 1 \leq x_1 \leq i-1 \rangle$	$\langle i \leq x_1 \leq i \rangle$	$\langle i+1 \leq x_1 \leq N \rangle$

The algorithm discovers that statement 2 kills statement 4 (type I), and statement 6 kills statements 2 and 4 (type II). The following temporal slices are solutions to the flow equations at the end of the loop:

	past	present	future
(2)	$\emptyset$	$\emptyset$	$\langle i+1 \leq x_1 \leq i+2 \rangle$
(4)	$\langle i \leq x_1 \leq i+1 \rangle$	$\langle i+2 \leq x_1 \leq i+2 \rangle$	$\langle i+3 \leq x_1 \leq N+2 \rangle$
(6)	$\langle 1 \leq x_1 \leq i-1 \rangle$	$\langle i \leq x_1 \leq i \rangle$	$\emptyset$

Intersection of these **past**, **present**, and **future** slices with **present**(1), **present**(3), **present**(5), and **present**(7), reveals the following dependences: statement 4 is anti-dependent on statement 7, statements 2 and 6 are anti-dependent on statement 1 (the anti-dependence to 6 is extraneous), statement 1 is flow-dependent on statement 4, and statement 3 is flow-dependent on statement 6.

4.2 Nested Loops

Since temporal slices describe the portion of an array assigned into relative to the immediate enclosing loop, the algorithm can be extended to nested loops by working from the inner-most to outermost loop. After completing the analysis of the innermost loop, compute the **past**, **present** and **future** for the next outer loop (leaving the inner loop indices unconstrained). The analysis is repeated for the immediately enclosing loop, as in the following example:

```
for i ← 1 to 10
    A[i,1] ← 0          /* 1 */
    for j ← 1 to 10
        A[i,j] ← j      /* 2 */
    endfor
endfor
```

After trivial analysis of the inner loop, the **past**, **present** and **future** temporal slices are computed for the outer loop (illustrated in Figure 10):

	past	present	future
(1)	$\langle 1 \leq x_1 \leq 1 \rangle$	$\langle 1 \leq x_1 \leq 1 \rangle$	$\langle 1 \leq x_1 \leq 1 \rangle$
(2)	$\langle 1 \leq x_2 \leq i-1 \rangle$	$\langle i \leq x_2 \leq i \rangle$	$\langle i+1 \leq x_2 \leq 10 \rangle$
(1)	$\langle 1 \leq x_1 \leq 10 \rangle$	$\langle 1 \leq x_1 \leq 10 \rangle$	$\langle 1 \leq x_1 \leq 10 \rangle$
(2)	$\langle 1 \leq x_2 \leq i-1 \rangle$	$\langle i \leq x_2 \leq i \rangle$	$\langle i+1 \leq x_2 \leq 10 \rangle$

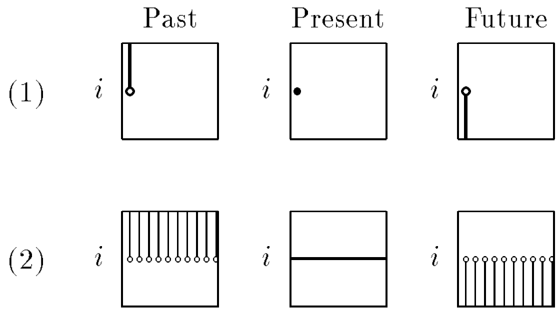

Figure 10: **past**, **present**, and **future** temporal slices input to algorithm **array_kill**.

In this example, since **future**(2) and **present**(2) contain elements defined in **present**(1), the assignment at (2) kills definitions of (1). This type II kill reduces **past**(1) to $\emptyset$, and restricts **future**(2) to $\langle 2 \le x_1 \le 10\rangle$, $\langle i+1 \le x_2 \le 10\rangle$.

5 Conclusion

For dependence analysis to be tractable, data dependences must be summarized for loops and subprograms. The Data Access Descriptor representation provides an efficient geometric summary of data dependences. Previously, however, the information in the Data Access Descriptor abstraction has only been regarded as a conservative approximation of data dependences, even though the information it contains is often precise.

By detecting when the dependences described by a Data Access Descriptor are exact, our work increases the number of analysis techniques based on them. The **union_exact** algorithm determines whether the union of exact simple sections in n dimensions remains exact, thereby enhancing the precision of Data Access Descriptors.

Data dependence analyses based on Data Access Descriptors are improved by methods for extracting array kill information (algorithm **array_kill**) and calculating direction vectors from the simple section (**temporal_slice** algorithm). This new precision and information is valuable in exposing optimization and parallelization opportunities previously inaccessible via Data Access Descriptors.

References

[Aho85] A. Aho, R. Sethi, J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1985.

[Bal89a] V. Balasundaram, K. Kennedy. *A Technique for Summarizing Data Access and Its Use in Parallelism Enhancing Transformations.* SIGPLAN, Portland, Oregon, June 1989.

[Bal89b] V. Balasundaram. *Interactive Parallelization of Numerical Scientific Programs.* Ph.D. Thesis, Department of Computer Science, Rice University, Houston, Texas, July 1989.

[Bal90] V. Balasundaram. *A Mechanism for Keeping Useful Internal Information in Parallel Programming Tools: The Data Access Descriptor.* Journal of Parallel and Distributed Computing, Vol. 9, pages 154-170, 1990.

[Ban88] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[Ban76] U. Banerjee. *Data Dependence in Ordinary Programs.* M.S. Thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, TR 76-837, November 1976.

[BuCy86] M. Burke, R. Cytron. *Interprocedural Dependence Analysis and Parallelization.* Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, pages 162-175, June 1986.

[Cal86] D. Callahan. *A Global Approach to Parallelism Detection.* PhD Thesis, Rice University, July 1986.

[GrSt90] T. Gross, P. Steenkiste. *Structured Dataflow Analysis for Arrays and its Use in an Optimizing Compiler.* Software—Practice and Experience, Vol. 20(2), pages 133-155, February 1990.

[Wol89] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, 1989.

Appendices

A Data Access Descriptors

In [Bal89b], the Data Access Descriptor is defined as follows:

> **Definition 1 (Data Access Descriptor)** Given an n dimensional array A that is accessed within a region R of a program, the Data Access Descriptor for A in R, denoted by $\delta_R(A)$ is a 3 tuple $\langle \theta | S | \tau \rangle$, where θ is the reference template, S is the simple section approximating the portion of A that is accessed within R, and τ is the traversal order of S.

The *reference template* of the Data Access Descriptor is used in translating the Data Access Descriptor between differing contexts. Access order of the array is described by the *traversal order*. Relevant to this paper, however, are Balasundaram's *simple sections*.

A.1 Simple Sections

The *simple section* is used to the section of an array that is accessed by with a region, i.e. loop or subprogram body. Simple sections are bounded by simple boundaries.

> **Definition 2 (simple boundary)** Given an n dimensional space with coordinate axes $x_1, x_2, \ldots, x_n$, a *simple boundary* is a hyperplane of the form $x_i = c$ or $x_i \pm x_j = c$, where x_i, x_j are any two different coordinate axes, and c is an integer constant.

We later extend this definition to allow symbolic integer variables to replace the integer constant. A simple section is described with simple boundaries.

> **Definition 3 (simple section)** Any convex polytope with simple boundaries is a *simple section*.

A boundary that does not contribute to the shape of the section being described is termed *redundant*. We also require all boundaries to be *tight*—they must contain at least one point of the simple section.

We follow Balasundaram's notation for describing a simple section S by its *simple boundary pairs*.

Definition 4 (simple boundary pair) A simple boundary pair is an inequality of the form $\alpha \leq \psi(x_1, \ldots, x_n) \leq \beta$, where $\psi(x_1, \ldots, x_n)$ is a function of the coordinate axes $x_1, \ldots, x_n$, such that $\psi(x_1, \ldots, x_n) = \alpha$ and $\psi(x_1, \ldots, x_n) = \beta$ are both simple boundaries of the section S. $\psi(x_1, \ldots, x_n) = \alpha$ is called a *lower boundary* and $\psi(x_1, \ldots, x_n) = \beta$ is called an *upper boundary*.

Figure 1 shows a two dimensional simple section along with functions defining the simple boundaries. Note that the upper bound $x - y = c_5$ is both redundant and tight.

Two or more simple sections may be compared for data conflict by performing an *intersection* operation, and summarized into one simple section by performing a *union* operation. If an upper bound on the number of array dimensions is fixed, these operations require constant time.

A.1.1 Intersection of simple sections

Computing the intersection of two simple sections, $S = S_1 \cap_{simp} S_2$, results in a simple section describing common array elements accessed by the references described by S_1 and S_2. It is important to note that *simple sections are closed under intersection*. This implies that the intersection of simple sections S_1 and S_2 is necessarily exact if S_1 and S_2 are themselves exact.

Given two simple sections S_1 and S_2, both of dimension n, the intersection S is computed by comparing corresponding boundary pairs from the two simple sections.

From the sets of simple boundaries, the intersection operation always chooses the "more interior" of the two boundary pairs to participate in the corresponding boundary pair of the resulting simple section S. An additional check ensures that $\emptyset$ is returned if S_1 and S_2 do not intersect.

In Figure 11, for example, the boundary β_1 is "more interior" to β_2 and is therefore chosen as a simple boundary of S. The lower bound α_2 is chosen similarly.

A.1.2 Union of simple sections

The union of two simple sections, $S = S_1 \cup_{simp} S_2$, is performed in a manner similar to intersection. The "more exterior" of the simple boundaries being compared is chosen to be a simple boundary in the resulting simple section. This is exemplified in Figure 11 where β_2 and α_1 are combined to form the boundary pair $\alpha_1 \leq y \leq \beta_2$ in $S = S_1 \cup_{simp} S_2$.

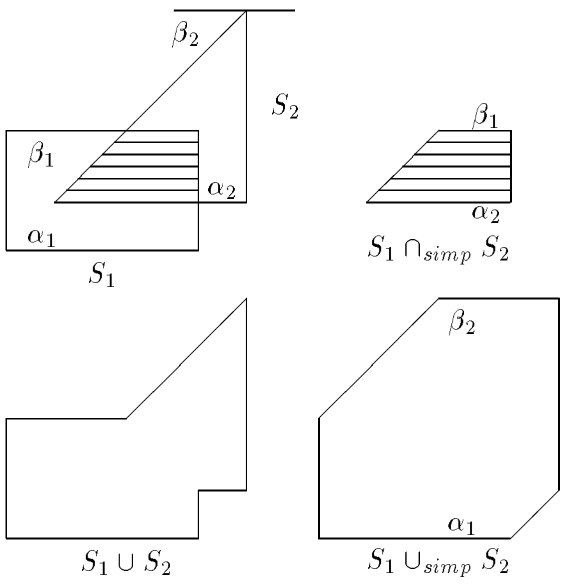

Figure 11: Intersection and union of simple sections. α and β represent lower and upper bounds of the boundary pair in the y dimension.

In [Bal89b] it is shown that $S_1 \cup_{simp} S_2$ is the smallest simple section containing the union $S_1 \cup S_2$. It is also proved that: $S_1 \cup_{simp} (S_2 \cup_{simp} S_3) = (S_1 \cup_{simp} S_2) \cup_{simp} S_3$.

Unlike intersection, *simple sections are* not *closed under set union*. Hence, the simple section representing the union of two simple sections, $S = S_1 \cup_{simp} S_2$, may be an approximation to $S_1 \cup S_2$. However, the union is in many cases exact, i.e. $S_1 \cup_{simp} S_2 = S_1 \cup S_2$, and therefore important as dependence and optimization information. Section 2 describes a test which determines if the resulting simple section S is exact.

B Proof of Exact Union Theorem

Definition 5 A simple boundary b_1 in an n dimensional simple section S is *adjacent* to a simple boundary $b_2 \in S$ if b_1 and b_2 form a 135° interior angle in the geometric section described by S.

Definition 6 An n dimensional union of simple sections, $S = S_1 \cup_{simp} S_2$, is *boundary consistent* with respect to S_1 and S_2 if for all $b \in S$ where b is the "exterior" boundary chosen from $b_1 \in S_1$, $b_2 \in S_2$:

$|b_1 - b_2| \neq 0$: A boundary b' adjacent to b in S_i must also be in S.

$|b_1 - b_2| = 0$: All corresponding simple sections in $n-1$ dimensional space induced in S_1 and S_2 by b_1 and b_2 respectively are exact under $\cup_{simp}$.

Theorem 2 The union of two simple sections, $S = S_1 \cup_{simp} S_2$, is *exact* if and only if S is boundary consistent.

Proof.

(1) ($\Longleftarrow$) Assume S is boundary consistent. We show that there is no point $x \in S$ such that $x \notin S_1$ and $x \notin S_2$. Assume $x \in S$ and $x \notin S_1$ and show $x \in S_2$. By the assumptions, x must lie outside of S_1 but in S. For this to happen, a simple boundary "exterior" to a boundary S_1 must be in S. This bound b must be from S_2. Since S is boundary consistent, b and its adjacent bounds define a section of S's perimeter. Since $x \in S$, x is bounded by b and boundaries adjacent to b. Continuing this argument by examining boundaries adjacent to b's boundaries, it is clear that $x \in S_2$. Hence $S = S_1 \cup_{simp} S_2$ is exact if S is boundary consistent.

(2) ($\Longrightarrow$) Assume $S = S_1 \cup_{simp} S_2$ exact. Proof by induction on the number of dimensions, n.

Base case. $n = 2$ Let S and S' describe two simple sections in 2 dimensional space.

$$
\begin{array}{llll}
 & S & & S' \\
B_1: & \alpha_1 \leq x_1 \leq \beta_1 & B'_1: & \alpha'_1 \leq x_1 \leq \beta'_1 \\
B_2: & \alpha_2 \leq x_2 \leq \beta_2 & B'_2: & \alpha'_2 \leq x_2 \leq \beta'_2 \\
B_3: & \alpha_3 \leq x_1 + x_2 \leq \beta_3 & B'_3: & \alpha'_3 \leq x_1 + x_2 \leq \beta'_3 \\
B_4: & \alpha_4 \leq x_1 - x_2 \leq \beta_4 & B'_4: & \alpha'_4 \leq x_1 - x_2 \leq \beta'_4
\end{array}
$$

Compare corresponding boundary pairs. Without loss of generality, consider the two cases when examining B_1 and B'_1:

$\beta_2 > \beta'_2$: Show that the adjacent boundaries must be equal to S's adjacent boundaries, B_3 and B_4, i.e. $\beta'_3 \leq \beta_3$ and $\beta'_4 \leq \beta_4$.
Suppose $\beta'_3 > \beta_3$. This implies that there exists at least one point in S' that is not in S and lies on the boundary β_3. Let x be the point of intersection of β'_3 and β_2. Then $\beta_2 > \beta'_2$ implies $x \notin S'$ and by assumption that $\beta'_3 > \beta_3$, $x \notin S$. An identical argument shows $\beta'_4 \leq \beta_4$.

$\beta_2 = \beta'_2$: In this case we must examine the simple sections induced by these boundaries in the 1-dimensional hyperplane. Since these simple sections are lines, they must be contiguous for the union to be exact. If the line segments are not contiguous, the points X creating the discontinuity will be in the union of S and S', but $X \not\subseteq S$ and $X \not\subseteq S'$.

Symmetric arguments can be made for B_i and B'_i, $2 \leq i \leq 4$. The above cases correspond to the conditions necessary for a simple section to be boundary consistent. Therefore, a simple section S in 2 dimensions is boundary consistent with respect to S_1 and S_2 if it is exact.

Induction Hypothesis. Assume simple section union in $n - 1$ dimensions is boundary consistent.

Induction Step. Show that the $S = S_1 \cup_{simp} S_2$ is boundary consistent if S is exact. Since a simple boundary can only lie in a plane defined by at most two coordinate axes (by the definition of simple boundary), we need only examine the $\frac{n(n-1)}{2}$ induced simple sections in 2 dimensions. By the induction hypothesis, these "projections" are boundary consistent. It follows that the n dimensional simple section is also boundary consistent because the simple bounds that comprise it, must be the simple bounds from the exact simple sections in 2 dimensions.

Combining (*1*) and (*2*) we have $S = S_1 \cup_{simp} S_2$ exact if and only if S is boundary consistent. $\square$