UMIACS-TR-93-69.1 July, 1993
CS-TR-3110.1

Lazy Array Data-Flow Dependence Analysis

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

Univ. of Maryland, College Park, MD 20742

Abstract

Automatic parallelization of real FORTRAN programs does not live up to users expectations yet, and dependence analysis algorithms which either produce too many false dependences or are too slow contribute significantly to this. In this paper we introduce data-flow dependence analysis algorithm which exactly computes value-based dependence relations for program fragments in which all subscripts, loop bounds and IF conditions are affine. Our algorithm also computes good affine approximations of dependence relations for non-affine program fragments. Actually, we do not know about any other algorithm which can compute better approximations.

And our algorithm is efficient too, because it is lazy. When searching for write statements that supply values used by a given read statement, it starts with statements which are lexicographically close to the read statement in iteration space. Then if some of the read statement instances are not "satisfied" with these close writes, the algorithm broadens its search scope by looking into more distant writes. The search scope keeps broadening until all read instances are satisfied or no write candidates are left.

We timed our algorithm on several benchmark programs and the timing results suggest that our algorithm is fast enough to be used in commercial compilers — it usually takes 5 to 15 percent of `f77 -O2` compilation time to analyze a program. Most programs in the 100-line range take less than 1 second to analyze on a SUN SparcStation IPX.

Appeared in the Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 1994, pp. 311–325.

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

Lazy Array Data-Flow Dependence Analysis *

Vadim Maslov
Department of Computer Science
University of Maryland, College Park, MD 20742
`vadik@cs.umd.edu`, (301)-405-2726

Abstract

Automatic parallelization of real FORTRAN programs does not live up to users expectations yet, and dependence analysis algorithms which either produce too many false dependences or are too slow contribute significantly to this. In this paper we introduce data-flow dependence analysis algorithm which exactly computes value-based dependence relations for program fragments in which all subscripts, loop bounds and IF conditions are affine. Our algorithm also computes good affine approximations of dependence relations for non-affine program fragments. Actually, we do not know about any other algorithm which can compute better approximations.

And our algorithm is efficient too, because it is lazy. When searching for write statements that supply values used by a given read statement, it starts with statements which are lexicographically close to the read statement in iteration space. Then if some of the read statement instances are not "satisfied" with these close writes, the algorithm broadens its search scope by looking into more distant writes. The search scope keeps broadening until all read instances are satisfied or no write candidates are left.

We timed our algorithm on several benchmark programs and the timing results suggest that our algorithm is fast enough to be used in commercial compilers — it usually takes 5 to 15 percent of `f77 -O2` compilation time to analyze a program. Most programs in the 100-line range take less than 1 second to analyze on a SUN SparcStation IPX.

*This work is supported by an NSF grant CCR-9157384 and by the Packard Foundation.

1 Introduction

Currently automatic parallelization of real-life FORTRAN programs is not as perfect as users desire. As recent studies [EHLP91, Blu92, May92] indicate, in many cases false dependences between statements introduced by inexact dependence analysis algorithms prevent loops from being parallelized. In the introduction we analyze the basic reasons for false dependences and show how the algorithm introduced in this paper avoids introducing false dependences without loosing efficiency.

Value-based dependences vs memory-based dependences. Traditionally dependence analyzers of parallelizing compilers and environments computed only *memory-based* dependences. That is, they reported that there is a dependence between two statements of a program if these statements access the same memory cell. For example, for the program in Figure 1(a) traditional dependence analyzer (for example, that of Parascope) reports that there is a flow dependence from statement S_0 to statement S_1 carried by the loop `rs`.

Since memory-based dependences often can be removed by program transformations such as array expansion and privatization (for example, array `XRSIQ` can be privatized in loop `rs`), recent research activity has focused on *value-based* (or *data-flow*) dependences, which need to be computed to perform these transformations. Value-based dependences, introduced by Feautrier in [Fea88b], reflect true flow of values in a program unobscured by details of storing data in memory.

Intuitively, a value-based dependence exists between two statement instances if there exists memory-based dependence between them and value written in the first statement instance is actually used in the second instance, that is, the memory cell written in the first statement instance is not overwritten before the second instance occurs.

```
  INTEGER rs, p, q, i
  DO rs = 1, nrs
    DO q = 1, np
      DO i = 1, mb
S0:     XRSIQ(i,q) = 0
      END DO
    END DO
    DO p = 1, np
      DO q = 1, p
        ...
        DO i = 1, mb
S1:       XRSIQ(i,q) = XRSIQ(i,q) + ...
S2:       XRSIQ(i,p) = XRSIQ(i,p) + ...
        END DO
      END DO
    END DO
    ...
  END DO
```

(a) Fragment of subroutine `OLDA` from TRFD

```
      DO i = 1, NMOL1
        DO j = i + 1, NMOL
C         Inlined subroutine CSHIFT
 S1:      XL(1) = XMA-XMB
 S2:      XL(2) = XMA-XB(1)
 S3:      XL(3) = XMA-XB(3)
...
S12:      XL(12) = XA(2)-XB(3)
S13:      XL(13) = XA(1)-XB(2)
S14:      XL(14) = XA(3)-XB(2)
          DO k = 1,14
S16:        XL(k) = XL(k) - ...
          END DO
          ...
        END DO
      END DO
```

(b) Fragment of subroutine `INTERF` from MDG

Figure 1: Examples from Perfect Club benchmark

Let's consider a program in Figure 1(a) which is slightly simplified fragment of the subroutine `OLDA` from the Perfect Club benchmark suite [B+89]. In it there exists loop-independent value-based dependence from statement S_0 to statement S_1 but no loop-carried value-based dependences from S_0 to S_1, because statement S_1 reads value of **`XRSIQ(i,q)`** written on the same iteration of the loop **`rs`** and does not read values written on previous iterations of loop **`rs`**.

Dependence Representation. Traditionally dependences were represented by direction vectors [Wol82] and dependence distances [Mur71]. Direction vectors represent a relationship between statement instances involved in dependence inexactly, and dependence distances are limited to representing only fixed differences between write and read variables. The *exact* relationship should be provided, if we want to use advanced loop transformation and code generation techniques such as [Fea92a, Fea92b, AL93, KP93]. Some array expansion and privatization algorithms also require exact dependence information [Fea88b].

Recently researchers started to use *source functions* to represent value-based dependences. For a given statement instance $S_2[\mathbf{r}]$ the source function produces coordinates of the statement instance $S_1[\mathbf{w}]$ such that $S_1[\mathbf{w}]$ supplies the value used in $S_2[\mathbf{r}]$. The version of source function computed in [Fea91] is called *Quasi-Affine Selection Tree (quast).* In [MAL93] a different term is used for the same object — *Last Write Tree (LWT).* For example, quast for the statement S_2 in Figure 1(a) is $\mathsf{Src}(S_2[p,q,i]) =$

$$\left[\begin{array}{llll} \texttt{if} & q=p & \texttt{then} & S_1[p,q,i] \\ \texttt{elseif} & q\geq 2 & \texttt{then} & S_2[p,q-1,i] \\ \texttt{else} & & \texttt{then} & S_0[p,i] \end{array}\right. .$$

We found that LWTs/quasts have several drawbacks as a method of dependence representation (see below), and we decided to use *dependence relations* introduced in [Pug91] to represent value-based dependences. If a pair consisting of the given instance of write statement $S_1[\mathbf{w}]$ and read statement $S_2[\mathbf{r}]$ belongs to the dependence relation then there is a value-based dependence from $S_1[\mathbf{w}]$ to $S_2[\mathbf{r}]$.

For example, the above LWT can be represented as a union of 3 simple relations:

$$\begin{array}{lll} S_1[p,q,i] & \rightarrow S_2[p,q,i] & \mid 1\leq p=q\leq \texttt{np} \wedge 1\leq i\leq \texttt{mb} \\ S_2[p,q-1,i] & \rightarrow S_2[p,q,i] & \mid 2\leq q<p\leq \texttt{np} \wedge 1\leq i\leq \texttt{mb} \\ S_0[p,i] & \rightarrow S_2[p,1,i] & \mid 2\leq p\leq \texttt{np} \wedge 1\leq i\leq \texttt{mb} \end{array} \quad (1)$$

Similarly, the source function for S_1 is:

$$\begin{array}{lll} S_2[p,q-1,i] & \rightarrow S_1[p,q,i] & \mid 2\leq p=q\leq \texttt{np} \wedge 1\leq i\leq \texttt{mb} \\ S_2[p-1,q,i] & \rightarrow S_1[p,q,i] & \mid 2\leq p\leq \texttt{np} \wedge q=p-1 \wedge 1\leq i\leq \texttt{mb} \\ S_1[p-1,q,i] & \rightarrow S_1[p,q,i] & \mid p\leq \texttt{np} \wedge 1\leq q\leq p-2 \wedge 1\leq i\leq \texttt{mb} \\ S_0[1,i] & \rightarrow S_1[1,1,i] & \mid 1\leq i\leq \texttt{mb} \end{array} \quad (2)$$

These two source functions may seem to be complicated, but if we draw dependence graph that they produce (see Figure 2, only axes p and q are shown), we will see that they encode elegant and relatively simple value flow pattern.

We think that dependence relations have the following advantages comparing to LWTs/quasts:

- If we want to know, under which condition a given LWT leaf is valid, we need to build and simplify a conjunction of conditions from nodes on the path from this leaf to the LWT root. Since conditions on the ELSE branches of the tree are negated, we end up having *disjunction of conjunctions* of constraints, which is much more difficult to handle than *conjunction* of constraints that we have in each simple relation of dependence relation.

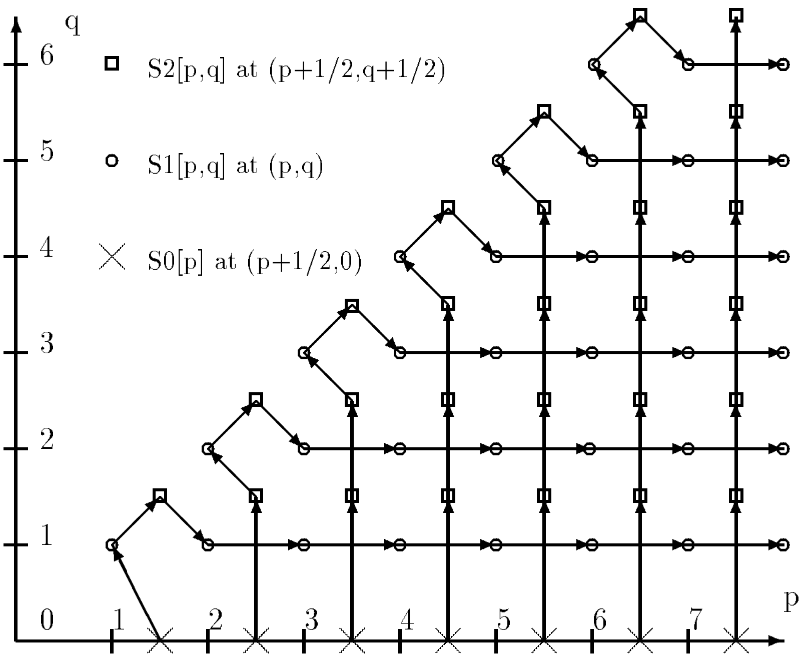

Figure 2: Dependence graph for the fragment of subroutine `OLDA` projected on p and q

- Integer division is used in quasts [Fea88a] to represent complicated dependence patterns (see example in Figure 6). It means that quast may contain non-affine functions. On the other hand, all expressions in the dependence relations are affine, because integer division is represented using wildcard variables.
- Computing affine approximations of non-affine dependence relations, we encounter a situation when a single read instance is dependent on several write instances (see Section 5) and therefore the relation between read and write instances is not a function anymore and can not be represented as LWT. However, it still can be represented as a dependence relation.

Computing value-based dependences efficiently. Computing value-based dependences is currently considered (by many people) to be too slow and inefficient to be used in production compilers. As we think one of the reasons of existing techniques inefficiency is that they treat all the statement instances that write to array in question as having the same chance to be source of a given read. However, since we are looking for a statement instance which *most recently* hit the memory location read by a read statement, we can expect that write statement instances which are lexicographically closer to the read in iteration space are more likely to be sources of read data.

Having this in mind, we decided to compute the source function for a given read statement starting with write statements which are lexicographically closer to this read statement, and then proceeding to the more and more distant statements, while keeping track of instances of read statement already covered by writes. When all read statement instances are covered, we can stop and not test for dependences from other writes to the read.

For example, let's consider a program in Figure 1(a). Using our algorithm, we are able to compute the dependence relations (1) and (2) not using information about references to `XRSIQ` other than in statements S_0, S_1, S_2. These other references do exist, and not having to prove that dependences from them to S_1 and S_2 are false dependences improves the performance of dependence analysis.

Or, let's consider another example in Figure 1(b) [1]. All instances of the read `XL(k)` from the statement S_{16} are covered with writes to `XL` from statements $S_1, \ldots, S_{14}$ as follows:

$$\begin{array}{ll} S_1[i,j] \rightarrow S_{16}[i,j,1] & \mid 1 \le i \le \mathtt{NMOL1} \wedge i{+}1 \le j \le \mathtt{NMOL} \\ \ldots & \ldots \\ S_{14}[i,j] \rightarrow S_{16}[i,j,14] & \mid 1 \le i \le \mathtt{NMOL1} \wedge i{+}1 \le j \le \mathtt{NMOL} \end{array} \quad (3)$$

We are able to prove this *not examining statement instances which precede* $S_1[i,j]$, that is, instances $S_1[i',j'], \ldots, S_{14}[i',j']$ such that $(i' < i) \vee (i' = i \wedge j' < j)$ and statements other than $S_1, \ldots, S_{14}$ referencing the array `XL`.

[1] In it IF statement with non-affine condition is removed from statement S_{16} to make the example more simple, however our techniques work even if this statement is present.

```
    DO i = 1, n
      DO j = 1, n
        x = F(i,j)
S0:     IF (x) THEN
S1:       A(j) = ...
        ELSE
S2:       A(j) = ...
        ENDIF
S3:     ... = A(j)
      END DO
    END DO
```

(a) Non-affine IF condition

```
    DO i = 1, n
      DO j = 1, n
        x = F(i,j)
S1:     A(x) = ...
S2:     ... = A(x)
      END DO
    END DO
```

(b) Non-affine subscript function

Figure 3: Non-affine program fragments

Handling non-affine conditions and subscripts. Let's consider a program fragment in Figure 3(a). In it the read `A(j)` from the statement S_3 is covered by the write `A(j)` from either statement S_1 or S_2 at the innermost loop level. All existing dependence analyzers (that we are aware of) can not recognize this because IF statement S_0 has non-affine condition `x = F(i,j)`. Not knowing that read `A(j)` is covered within the body of loop `j`, existing systems assume that there exist a flow dependence from S_1 and S_2 to S_3 carried by the loop `i` and therefore they can not parallelize loop `i`.

Non-affine subscript functions also confuse many existing systems. For fragment in Figure 3(b) they can not establish that the write to `A(x)` in statement S_1 completely covers the read `A(x)` in S_2 at the innermost level. As before, it happens because `x = F(i,j)` is non-affine function. Assuming that loop-carried dependence from S_1 to S_2 exist they can not parallelize loops `i` and `j`.

We can prove that both dependences are loop-independent by using techniques described in Section 5.

2 Definitions

Notation used is summarized in Figure 4.

Domain of our work. Our data dependence testing algorithm was originally designed to compute value-based dependences for *affine* program fragments. Affine program fragment is a loop nest such that in every statement of it all (1) subscript functions, (2) conditions in IF statements, and (3) loop bounds are affine functions of loop variables and symbolic constants. If either of these requirements is not satisfied, this fragment is *non-affine.*

Then we modified the algorithm to handle non-affine fragments (see Section 5), so actually it computes dependence relations for any structured program fragment which does not contain GOTO, BREAK and WHILE statements. Allowed are assignment statement, structured IF and FORTRAN-like DO loop.

Vectors and Statement Instances. *Vector* (also called *tuple*) is simply an ordered set of integers. Vectors are denoted with bold letters, such as $\mathbf{w}, \mathbf{r}, \mathbf{s}$. They are used to represent points in n-dimensional space.

The smallest unit of computation we consider in this paper is *statement instance.* The statement instance $W[\mathbf{w}, \mathbf{s}]$ is specified by W — statement of the program, $\mathbf{w}$ — vector of loop variables values (loops which surround the statement W are included), and by $\mathbf{s}$ — vector of symbolic constants.

We call variable a *symbolic constant* if it is not a loop variable and it is not assigned in the fragment of the program that we analyze. For starters we assume that the fragment being analyzed is the whole body of the procedure being analyzed, but later (in Section 5) we will see that the scope of our dependence analysis algorithm is dynamic and so is the definition of symbolic constant.

Sequencing predicate. We say that instance of statement W specified by loop variables vector $\mathbf{w}$ and symbolic constants vector $\mathbf{s}$ is *executed before* instance of statement R specified by loop variables vector $\mathbf{r}$ and symbolic constants vector $\mathbf{s}$ or $W[\mathbf{w}, \mathbf{s}] \ll R[\mathbf{r}, \mathbf{s}]$ iff

$$\mathbf{w}[1..n] \ll \mathbf{r}[1..n] \ \vee \ \mathbf{w}[1..n] = \mathbf{r}[1..n] \ \wedge \ W \ll R$$

where n is number of common loops surrounding both statements W and R.

Relations. *Relation* is a set of ordered pairs of vectors. $(\mathbf{w} \rightarrow \mathbf{r}) \in R$ means that pair $(\mathbf{w}, \mathbf{r})$ belongs to the relation R.

Since statement instance (which is elemental unit of computation for us) is specified not just by vector of integers, but by statement and vector of integers, we consider relations between statement instances. So we write $(W[\mathbf{w}] \rightarrow R[\mathbf{r}]) \in R$ when pair $(W[\mathbf{w}] \rightarrow R[\mathbf{r}])$ belongs to the relation R.

Operations on sets and relations that we use are summarized in Figure 5.

W, R	Specific statement of a program.
$R.A, W.B$	Specific read/write array reference A/B in a statement R/W.
$\mathsf{Arr}(A)$	Array or scalar variable referenced in a reference A.
$\mathbf{w}, \mathbf{r}$	An iteration space vector that represents a specific set of values of the loop variables. The individual values of the loop variables are referred to as $w_1, w_2, \ldots, r_1, r_2, \ldots$
$\mathbf{r}[x..y]$	Subvector of vector $\mathbf{r}$, consisting of components $r_x, r_{x+1}, ..., r_{y-1}, r_y$.
$\lvert\mathbf{w}\rvert$	Number of components in vector $\mathbf{w}$. For example, $\lvert\mathbf{r}[x..y]\rvert = y - x + 1$.
$[R, \mathbf{s}]$	The set of iteration vectors for which statement R is executed given symbolic constant vector $\mathbf{s}$.
$R.\mathbf{A}(\mathbf{r}, \mathbf{s})$	The vector of integers produced by subscript function of array reference $R.A$, when the loop variables are specified by $\mathbf{r}$, and symbolic constants are specified by $\mathbf{s}$.
$W \ll R$	Statement W occurs before statement R in a text of a program.
$\mathbf{w} \ll \mathbf{r}$	Vector $\mathbf{w}$ is lexicographically less than vector $\mathbf{r}$. That is, $(w_1 < r_1) \vee (w_1 = r_1 \wedge w_2 < r_2) \vee (w_1 = r_1 \wedge w_2 = r_2 \wedge w_3 < r_3) \vee \cdots$.
$W[\mathbf{w}, \mathbf{s}]$	An instance of statement W specified by the loop variables vector $\mathbf{w}$ and symbolic constants vector $\mathbf{s}$.
$W[\mathbf{w}, \mathbf{s}] \ll R[\mathbf{r}, \mathbf{s}]$	Statement instance $W[\mathbf{w}, \mathbf{s}]$ is executed before statement instance $R[\mathbf{r}, \mathbf{s}]$.

Figure 4: Notation used in this paper

Operation	Description	Definition
$\mathsf{domain}(F)$	The domain of the relation F	$x \in \mathsf{domain}(F) \Leftrightarrow \exists y \ s.t.\ (x \rightarrow y) \in F$
$\mathsf{range}(F)$	The range of the relation F	$y \in \mathsf{range}(F) \Leftrightarrow \exists x \ s.t.\ (x \rightarrow y) \in F$
$F \setminus S$	Restrict relation F to domain S	$(x \rightarrow y) \in F \setminus S \ \Leftrightarrow\ (x \rightarrow y) \in F \wedge x \in S$
F / S	Restrict relation F to range S	$(x \rightarrow y) \in F / S \ \Leftrightarrow\ (x \rightarrow y) \in F \wedge y \in S$
$\pi_{\mathbf{x}}(P(\mathbf{x}, \mathbf{y}))$	Project problem P on variables $\mathbf{x}$	$\{\mathbf{x} \mid \exists \mathbf{y} \ s.t.\ P(\mathbf{x}, \mathbf{y})\}$
$\pi_{\neg\mathbf{x}}(P(\mathbf{x}, \mathbf{y}))$	Project problem P on variables other than $\mathbf{x}$	$\pi_{\mathbf{y}}(P(\mathbf{x}, \mathbf{y}))$

Figure 5: Operations on vectors and relations

Value-based dependence definition and representation. The value-based dependence relation $DepRel$ that describes the dependences coming to the read reference $R.A$ of statement R is defined by the following:

$$\begin{gathered} \forall \mathbf{r}, \mathbf{s} : (V[\mathbf{v}, \mathbf{s}] \rightarrow R.A[\mathbf{r}, \mathbf{s}]) \in DepRel(\mathbf{w}, \mathbf{r}, \mathbf{s}) \Leftrightarrow \\ V[\mathbf{v}, \mathbf{s}] = \max_{\ll}(W[\mathbf{w}, \mathbf{s}] \mid \ \mathbf{w} \in [W, \mathbf{s}] \ \wedge \ \mathbf{r} \in [R, \mathbf{s}] \wedge \\ \mathsf{Arr}(W.B) = \mathsf{Arr}(R.A) \ \wedge \ W.\mathbf{B}(\mathbf{w}, \mathbf{s}) = R.\mathbf{A}(\mathbf{r}, \mathbf{s}) \wedge \\ W[\mathbf{w}, \mathbf{s}] \ll R[\mathbf{r}, \mathbf{s}]) \end{gathered} \tag{4}$$

This definition is constructive, that is, we can use it to actually compute the dependence relations. When lexicographical maximum is computed, the result is a dependence relation which is represented as a union of the following m simple dependence relations:

$$DepRel = \left[\begin{array}{l} W_1[\mathbf{w}, \mathbf{s}] \ \rightarrow R.A[\mathbf{r}, \mathbf{s}] \mid \ DepRel_1(\mathbf{w}, \mathbf{r}, \mathbf{s}) \\ \ldots \\ W_m[\mathbf{w}, \mathbf{s}] \rightarrow R.A[\mathbf{r}, \mathbf{s}] \mid \ DepRel_m(\mathbf{w}, \mathbf{r}, \mathbf{s}) \end{array} \right.$$

where each $DepRel_i$ is a conjunction of constraints and $\bigcup_{i=1}^{m} \pi_{\mathbf{r}, \mathbf{s}}(DepRel_i(\mathbf{w}, \mathbf{r}, \mathbf{s})) \subseteq [R, \mathbf{s}]$.

Since source functions may involve integer division by constant [Fea88a] and we want to keep conjuncts $DepRel_i$ affine, we use wild-card variables to represent the integer division. That is, we replace constraint $i = \lfloor k/c \rfloor$ with affine constraint $ci + \alpha = k \wedge 0 \leq \alpha \leq c - 1$.

Let's consider a program in Figure 6 [Fea88a] as an example of representing relatively complex dependence with dependence relations. Source function for the statement S_2 is defined as: $\mathsf{Src}(S_2) =$

$$\max_{\ll}(S_1[i, j] \mid 0 \leq i \leq M \ \wedge \ 0 \leq j \leq N \ \wedge \ 2i + j = k). \tag{5}$$

The PIP algorithm simplifies this to the quast in the right column of Figure 6. Our algorithm for computing lexicographical maximum (see Section A.1) simplifies (5) to the dependence relation:

$$\begin{array}{l} S_1[M, k{-}2M] \rightarrow S_2 \mid 2M \leq k \leq 2M{+}N \ \wedge \ M \geq 0 \\ S_1[i, k{-}2i] \rightarrow S_2 \mid k{-}1 \leq 2i \leq k \ \wedge \ k{-}N \leq 2i \ \wedge \ 0 \leq i \leq M \end{array} \tag{6}$$

In the 2nd conjunct of this relation i is not expressed as a function of read variables and symbolic constants, even though it is a function of them. There was some discussion among researchers as to whether dependence relations should contain explicit functional binding between read and write variables. We do not feel that this is necessary, but our algorithm can be modified to produce such binding. The above dependence relation expressed in the functional form but without use of integer division is:

$$\begin{array}{l} S_1[M, k{-}2M] \rightarrow S_2 \mid 2M \leq k \leq 2M{+}N \ \wedge \ M \geq 0 \\ S_1[i, k{-}2i] \rightarrow S_2 \mid \\ \quad 2i{+}\alpha = k \ \wedge \ 0 \leq \alpha \leq 1 \ \wedge \ 0 \leq k \leq 2M + 1 \ \wedge \ N \geq 1 \\ S_1[i, 0] \rightarrow S_2 \mid \\ \quad 2i{+}\alpha = k \ \wedge \ 0 \leq \alpha \leq 1 \ \wedge \ 0 \leq k = 2\delta \leq 2M \ \wedge \ N = 0 \end{array}$$

```
      DO i = 0, M
        DO j = 0, N
S1:       A(2*i+j) = ...
        END DO
      END DO
S2:   ... = A(k)
```

$$\mathsf{Src}(S_2) = \left[\begin{array}{llll} \texttt{if} & (-k+2M+N \geq 0) & & \\ \texttt{then} & \texttt{if} & (k-2M \geq 0) & \\ & \texttt{then} & S_1[M, k-2M] & \\ & \texttt{else} & \texttt{if} & ((-k+N+2(k \div 2)) \geq 0) \\ & & \texttt{then} & S_1[k \div 2, k-2(k \div 2)] \\ & & \texttt{else} & \perp \\ \texttt{else} & \perp & & \end{array}\right.$$

Figure 6: Program and source function represented as a quast

3 Machinery used

We use the *Disjunctive Normal Form* (DNF) to represent sets of vectors. The DNF is a disjunction of conjunctions of constraints which maps integer vectors to boolean values. Each constraint is an affine equality or inequality. DNF representing a set of vectors produces True for the vector that belongs to the set, and False otherwise.

We use the following basic operations on DNFs: $\wedge$, $\vee$, $\neg$, π, $\mathsf{RelMax1}_{\ll}$, $\mathsf{RelMax2}_{\ll}$. They can be broken into 3 classes.

Conjunct to conjunct: $\wedge$, π. We use the Omega test [Pug92] to simplify conjunctions of constraints and to prove that they have no solutions. The Omega test always performs the exact simplification.

Another useful operation performed by the Omega test that we use is *projection*. Projection of a conjunct $P(\mathbf{x}, \mathbf{y})$ on variables $\mathbf{x}$ is $\pi_{\mathbf{x}}(P(\mathbf{x}, \mathbf{y})) = \pi_{\neg\mathbf{y}}(P(\mathbf{x}, \mathbf{y})) = \{\mathbf{x} \mid \exists \mathbf{y}\ s.t.\ P(\mathbf{x}, \mathbf{y})\}$.

DNF to DNF: $\vee$, $\neg$. The Omega test works only with separate conjuncts. To allow the use of $\vee$ and $\neg$ operations, we implemented the DNF package on top of the Omega test. In it we always maintain the disjunctive normal form of the formula using distributive properties of operations $\wedge$ and $\vee$. To avoid combinatorial explosion when computing negation we used gist operation in a way proposed in [PW93a].

Lexicographical maximum: $\mathsf{RelMax}_{\ll}$. The function $\mathsf{RelMax1}_{\ll}$ (see Appendix A.1) computes the lexicographical maximum of the set of vectors $\mathbf{w}$ which is described by DNF $p(\mathbf{w}, \mathbf{r})$, where $\mathbf{r}$ is a vector of parameter variables. The function produces the DNF that binds maximized $\mathbf{w}$ with $\mathbf{r}$: $P_m(\mathbf{v}, \mathbf{r}) = \mathsf{RelMax1}_{\ll}(\mathbf{w} \mid p(\mathbf{w}, \mathbf{r}))$. It is defined as

$$\forall \mathbf{v}, \mathbf{r} : P_m(\mathbf{v}, \mathbf{r}) \Leftrightarrow \mathbf{v} = \max_{\ll}(\mathbf{w} \mid p(\mathbf{w}, \mathbf{r})).$$

The version of this function for a single conjunct is called $\mathsf{ProblemMax}_{\ll}$.

The function $\mathsf{RelMax2}_{\ll}$ (see Appendix A.2) computes the lexicographical maximum of two parametrized source functions. Given the source functions $R_1 = \{W_1[\mathbf{w}_1, \mathbf{s}] \to R[\mathbf{r}, \mathbf{s}] \mid C_1(\mathbf{w}_1, \mathbf{r}, \mathbf{s})\}$ and $R_2 = \{W_2[\mathbf{w}_2, \mathbf{s}] \to R[\mathbf{r}, \mathbf{s}] \mid C_2(\mathbf{w}_2, \mathbf{r}, \mathbf{s})\}$ it produces the relation $R_m = \mathsf{RelMax2}_{\ll}(R_1, R_2)$ which is defined as

$$\forall \mathbf{w}, \mathbf{r}, \mathbf{s} : (W[\mathbf{w}, \mathbf{s}] \to R[\mathbf{r}, \mathbf{s}]) \in R_m \Leftrightarrow$$
$$W[\mathbf{w}, \mathbf{s}] = \max_{\ll}(R_1^{-1}(\mathbf{r}, \mathbf{s}), R_2^{-1}(\mathbf{r}, \mathbf{s}))$$

Related work. Feautrier developed the PIP algorithm (an integer version of simplex algorithm) [Fea88a] to compute an equivalent of $\mathsf{ProblemMax}_{\ll}$. We are not aware of any performance figures for the PIP, and Figure 8 suggests that it is slow. His analogue of $\mathsf{RelMax2}_{\ll}$ does not simplify the resulting quasts, so they may become very big. To simplify quasts one needs to perform *negation* and it is not mentioned in Feautrier's papers as far as we know.

Pugh and Wonnacott in [PW93a] advocate the use of the Presburger arithmetic subclass for dependence testing. We think that their subclass is equivalent to the class of formulas that can be built using operations listed in this section.

4 Lazy dependence analysis

In Figure 7 we present the algorithm which computes value-based dependences for a given read reference. Dependence graph for the whole program is built by applying the algorithm to every read reference of every statement.

Our algorithm can be viewed as a lazy implementation of the definition (4). Let's consider a set of read statements instances $R[\mathbf{r}, \mathbf{s}]$ for which we are computing source function. The set of all candidate write instances $\{W[\mathbf{w}, \mathbf{s}] \mid W[\mathbf{w}, \mathbf{s}] \ll R[\mathbf{r}, \mathbf{s}]\}$ is broken into n convex subsets $\omega_i(\mathbf{r}, \mathbf{s})$ such that for any $\mathbf{r}, \mathbf{s}$ such that $R[\mathbf{r}, \mathbf{s}]$ is executed

$$\omega_n(\mathbf{r}, \mathbf{s}) \ll \cdots \ll \omega_2(\mathbf{r}, \mathbf{s}) \ll \omega_1(\mathbf{r}, \mathbf{s}) \ll R[\mathbf{r}, \mathbf{s}]$$

These subsets are created on the fly as me move from the write instances $W[\mathbf{w}, \mathbf{s}]$ that are lexicographically close to the read instances $R[\mathbf{r}, \mathbf{s}]$ to the more distant write instances (lines 10–12 and 30–44 of the algorithm).

1: **INPUT**: $R.A$: read reference surrounded by n loops with variables $\mathbf{r} = (r_1, ..., r_n)$.
$\mathbf{s}$ is a vector of symbolic constants.
2: **OUTPUT**: Dependence relation for the read reference $R.A$.
That is, $\{W[\mathbf{v},\mathbf{s}] \rightarrow R[\mathbf{r},\mathbf{s}]\} \in \mathit{DepRel} \;\Leftrightarrow\; W[\mathbf{v},\mathbf{s}] = \max_{\ll}(\, W[\mathbf{w},\mathbf{s}] \mid \mathbf{w} \in [W,\mathbf{s}] \wedge \mathbf{r} \in [R,\mathbf{s}] \wedge$
$\mathsf{Arr}(W.B) = \mathsf{Arr}(R.A) \wedge W.\mathbf{B}(\mathbf{w},\mathbf{s}) = R.\mathbf{A}(\mathbf{r},\mathbf{s}) \wedge W[\mathbf{w},\mathbf{s}] \ll R[\mathbf{r},\mathbf{s}]\,)$
4: Relation $\mathit{DepRel} := \{\emptyset\}$; Relation WrMax
5: Dnf $\mathit{NotCovered}(\mathbf{r},\mathbf{s}) := \mathsf{IsExecuted}(R[\mathbf{r},\mathbf{s}])$
6: Integer $\mathit{FixLoops} := n$
7: Statement $W := R$
8: Boolean $\mathit{SingleWrite}$:= True; Boolean $\mathit{LessFlag}$:= False
10:While ($\mathit{NotCovered}$ is feasible) do
11: W := statement preceding statement W
12: Statement W is surrounded by m loops with variables $\mathbf{w} = (w_1, ..., w_m)$
13: (∗ Here unfixed zone consists of loops with depths from $\mathit{FixLoops} + 1$ to n. ∗)
15: If (W is assignment statement and it writes to $\mathsf{Arr}(R.A)$) then
16: (∗ Find source function for instances of reference $R.A[\mathbf{r},\mathbf{s}]$ which are $\mathit{NotCovered}(\mathbf{r},\mathbf{s})$ ∗)
17: Dnf $\mathit{SameCell}(\mathbf{w},\mathbf{r},\mathbf{s}) := \mathit{NotCovered}(\mathbf{r},\mathbf{s}) \wedge R.\mathbf{A}(\mathbf{r},\mathbf{s}) = W.\mathbf{B}(\mathbf{w},\mathbf{s}) \wedge \mathsf{IsExecuted}(W[\mathbf{w},\mathbf{s}])$
18: Conjunct $\mathit{Wsub}(\mathbf{w},\mathbf{r}) := \mathbf{w}[1..\mathit{FixLoops}] = \mathbf{r}[1..\mathit{FixLoops}] \wedge (\mathit{LessFlag} \Rightarrow w_{\mathit{FixLoops}+1} < r_{\mathit{FixLoops}+1})$
19: Dnf $\mathit{DepProb}(\mathbf{w},\mathbf{r},\mathbf{s}) := \mathit{SameCell}(\mathbf{w},\mathbf{r},\mathbf{s}) \wedge \mathit{Wsub}(\mathbf{w},\mathbf{r})$
20: Relation $\mathit{Cmax} := \mathsf{RelMax1}_{\ll}(W[\mathbf{w},\mathbf{s}] \rightarrow R.A[\mathbf{r},\mathbf{s}] \mid \mathit{DepProb}(\mathbf{w},\mathbf{r},\mathbf{s}))$
22: If ($\mathit{SingleWrite}$) then
23: $\mathit{DepRel} := \mathit{DepRel} \cup \mathit{Cmax}$
24: $\mathit{NotCovered} := \mathit{NotCovered} \wedge \neg\mathsf{range}(\mathit{Cmax})$
25: Else
26: $\mathit{WrMax} := \mathsf{RelMax2}_{\ll}(\mathit{WrMax}, \mathit{Cmax})$
27: EndIf
30: ElseIf (statement W is `EndDo` or `Do i=`) then (∗ Enter loop body through its end ∗)
31: If ($\mathit{SingleWrite}$) then
32: If (statement W is `Do i=`) then
33: $\mathit{FixLoops} := \mathit{FixLoops} - 1$; $\mathit{LessFlag}$:= True
34: W := `EndDo` stmt for loop with header W
35: Else (∗ statement W is `EndDo` ∗)
36: $\mathit{LessFlag}$:= False
37: EndIf
38: $\mathit{WrMax} := \{\emptyset\}$; $\mathit{SingleWrite}$:= False
39: $\mathit{StopLoop}$:= `Do i=` stmt of the loop whose `EndDo` stmt is W
40: ElseIf ($\neg\mathit{SingleWrite} \wedge W = \mathit{StopLoop}$) then
41: $\mathit{DepRel} := \mathit{DepRel} \cup \mathit{WrMax}$
42: $\mathit{NotCovered} := \mathit{NotCovered} \wedge \neg\mathsf{range}(\mathit{WrMax})$
43: $\mathit{SingleWrite}$:= True
44: EndIf
50: ElseIf (statement W is entry to the subroutine) then
51: $\mathit{DepRel} := \mathit{DepRel} \cup \{\mathit{Entry} \rightarrow R.A[\mathbf{r},\mathbf{s}] \mid \mathit{NotCovered}(\mathbf{r},\mathbf{s})\}$
52: Break out of While loop 10
55: ElseIf (statement W is `EndIf` or `Else` or `If (...) then`) then
56: (∗ Do nothing ∗)
60: EndIf
61:EndDo
62:Return (DepRel)

Figure 7: Value-based dependence analysis algorithm

Each of the subsets $\omega_i(\mathbf{r},\mathbf{s})$ can include instances of one or more write statements. If boolean flag *SingleWrite* is True, then current $\omega_i(\mathbf{r},\mathbf{s})$ includes instances of only one statement W and to get dependences from $W[\mathbf{w},\mathbf{s}]$ to $R[\mathbf{r},\mathbf{s}]$ we need simply to compute $\max_{\ll}$ of eligible instances of W (lines 20–23). If *SingleWrite* is False, then instances of several statements can be present at ω_i and after finding source function for each statement (line 20) we have to compute a maximum of these source functions (line 26).

After examining a statement we move to a preceding statement. If we reach beginning of the loop L which surrounds the read statement R where we have started, we move to the end of this loop (line 34), unfix the loop L, require to consider the writes only from previous iterations of L (lines 33 and 18), and enter multiple-write-statement zone (line 38). When later we reach beginning of loop L, the multiple-write-statement zone is over and we add lexicographical maximum of the source functions computed in this zone to the resulting dependence relation (line 41).

Lines 17–20 implement the value-based dependence definition (4). Line 17 selects writes that are executed and that hit the same memory locations as reads. Line 18 selects writes which belong to the current ω_i. The function IsExecuted returns conjunct that describes conditions under which the given statement executes. This conjunct consists of conditions imposed on loop variables by loop bounds and IF statements surrounding the statement. In other words, $\mathsf{IsExecuted}(R[\mathbf{r},\mathbf{s}]) \Leftrightarrow R[\mathbf{r},\mathbf{s}] \in [R,\mathbf{s}]$.

The subexpression $\mathsf{range}(Cmax)$ in lines 24 and 42 describes a set of read instances that has been covered by writes from the current ω_i. Remaining not covered reads are described by the problem *NotCovered* which is initialized in line 5 and is updated in lines 24 and 42.

Termination. Termination of the algorithm is proven trivially. The set of vectors specified by the problem *NotCovered* becomes smaller or remains the same with each iteration of the loop 10–61. When *NotCovered* becomes empty algorithm stops and the resulting relation is returned (lines 10 and 62). If this does not happen then we reach the beginning of the program fragment being examined (line 50). We can have some not covered reads left, and we let them to depend on the *Entry* node (line 51).

Computational complexity. Worst-case computational complexity of the algorithm in the number of calls to RelMax1 and RelMax2 is $O(nd)$, where n is number of statements writing to $\mathsf{Arr}(R.A)$ and d is number of loops around statement R. Since usually $d \leq 5$, we can state that worst-case time complexity is $O(n)$. Practical complexity is lower, since usually read instances are covered after visiting only small number of candidate writes.

Each call to RelMax1 and RelMax2 is in the worst case NP-complete in the number of integer programming problems to be solved. In practice, however, only small number of problems is solved in each call.

4.1 Example of the algorithm work

Here we demonstrate how our algorithm computes the source function for statement S_1 of subroutine `OLDA` (see Figure $1(a)$).

First, for each write–read pair we summarize all constraints on loop variables and symbolic constants except for ordering constraints:

$C_0 : S_0 \to S_1$	$C_1 : S_1 \to S_1$	$C_2 : S_2 \to S_1$
$q_w = q_r$	$q_w = q_r$	$p_w = q_r$
$i_w = i_r$	$i_w = i_r$	$i_w = i_r$
$1 \leq q_r \leq p_r$	$1 \leq q_r \leq p_w, p_r$	$1 \leq q_w \leq q_r \leq p_r$
$p_r \leq \mathtt{np}$	$p_w, p_r \leq \mathtt{np}$	$p_r \leq \mathtt{np}$
$1 \leq i_r \leq \mathtt{mb}$	$1 \leq i_r \leq \mathtt{mb}$	$1 \leq i_r \leq \mathtt{mb}$

Then we take care about ordering constraints. The algorithm breaks a set of write statement instances into a sum of disjoint subsets $\omega_2(\mathbf{r}), \ldots, \omega_{ns}(\mathbf{r})$ such that for any $\mathbf{r}$ such that $R[\mathbf{r}]$ is executed: $\omega_{ns}(\mathbf{r}) \ll \ldots \ll \omega_2(\mathbf{r}) \ll R[\mathbf{r}]$. For the read instances $S_1[rs_r, p_r, q_r, i_r] \mid 1 \leq rs_r \leq \mathtt{nrs} \wedge 1 \leq q_r \leq p_r \leq \mathtt{np} \wedge 1 \leq i_r \leq \mathtt{mb}$ these subsets are the following:

$$
\begin{aligned}
\omega_2 &= S_2[rs_r, p_r, q_r, i_w] \mid 1 \leq i_w < i_r \\
&\quad\; S_1[rs_r, p_r, q_r, i_w] \mid 1 \leq i_w < i_r \\
\omega_3 &= S_2[rs_r, p_r, q_w, i_w] \mid 1 \leq q_w < q_r \;\wedge\; 1 \leq i_w \leq \mathtt{mb} \\
&\quad\; S_1[rs_r, p_r, q_w, i_w] \mid 1 \leq q_w < q_r \;\wedge\; 1 \leq i_w \leq \mathtt{mb} \\
\omega_4 &= S_2[rs_r, p_w, q_w, i_w] \mid 1 \leq q_w \leq p_w < p_r \;\wedge\; 1 \leq i_w \leq \mathtt{mb} \\
&\quad\; S_1[rs_r, p_w, q_w, i_w] \mid 1 \leq q_w \leq p_w < p_r \;\wedge\; 1 \leq i_w \leq \mathtt{mb} \\
\omega_5 &= S_0[rs_r, q_w, i_w] \mid 1 \leq q_w \leq \mathtt{np} \;\wedge\; 1 \leq i_w \leq \mathtt{mb} \\
\omega_6 &= S_2[rs_w, p_w, q_w, i_w] \mid 1 \leq rs_w < rs_r \;\wedge \\
&\qquad 1 \leq q_w \leq p_w \leq \mathtt{np} \;\wedge\; 1 \leq i_w \leq \mathtt{mb} \\
&\quad\; S_1[rs_w, p_w, q_w, i_w] \mid 1 \leq rs_w < rs_r \;\wedge \\
&\qquad 1 \leq q_w \leq p_w \leq \mathtt{np} \;\wedge\; 1 \leq i_w \leq \mathtt{mb} \\
&\quad\; S_0[rs_w, q_w, i_w] \mid 1 \leq rs_w < rs_r \;\wedge \\
&\qquad 1 \leq q_w \leq \mathtt{np} \;\wedge\; 1 \leq i_w \leq \mathtt{mb}
\end{aligned}
$$

Now we start moving from ω_2 back in space/time keeping track of covered S_1 instances. We don't mention constraints on `rs` for brevity and because this variable does not appear in subscript functions. Initially $NotCovered(\mathbf{r},\mathbf{s}) = (1 \leq q_r \leq p_r \leq \mathtt{np} \;\wedge\; 1 \leq i_r \leq \mathtt{mb})$.

ω_2: $\omega_2 \wedge C_1$ and $\omega_2 \wedge C_2$ have no solutions. So ω_2 doesn't contribute to dependence.

ω_3: $C_1 \wedge \omega_3$ is not feasible, but $C_2 \wedge \omega_3 = (1 \leq q_w < p_w = q_r = p_r \leq \mathtt{np} \;\wedge\; 1 \leq i_w = i_r \leq \mathtt{mb})$. Computing $\mathsf{RelMax1}_{\ll}(S_2[p_w, q_w, i_w] \to S_1[p_r, q_r, i_r] \mid C_2 \wedge \omega_3)$ we get

$$
\begin{gathered}
S_2[p_r, q_r - 1, i_r] \to S_1[p_r, q_r, i_r] \mid \\
2 \leq p_r = q_r \leq \mathtt{np} \;\wedge\; 1 \leq i_r \leq \mathtt{mb}
\end{gathered}
$$

Now we cover area $2 \le p_r = q_r \le \mathtt{np} \wedge 1 \le i_r \le \mathtt{mb}$ and therefore $NotCovered = (p_r = q_r = 1 \wedge 1 \le i_r \le \mathtt{mb}) \vee (1 \le q_r < p_r \le \mathtt{np} \wedge 1 \le i_r \le \mathtt{mb})$.

ω_4: $(C_2 \wedge \omega_4 \wedge NotCovered) = (1 \le q_w \le p_w = q_r < p_r \le \mathtt{np} \wedge 1 \le i_w = i_r \le \mathtt{mb})$. Maximum of this is $S_2[q_r, q_r, i_r] \mid 1 \le q_r < p_r \le \mathtt{np} \wedge 1 \le i_r \le \mathtt{mb}$.
$(C_1 \wedge \omega_4 \wedge NotCovered) = (1 \le q_w = q_r \le p_w < p_r \le \mathtt{np} \wedge 1 \le i_w = i_r \le \mathtt{mb})$ leading to maximum $S_1[p_r - 1, q_r, i_r] \mid 1 \le q_r < p_r \le \mathtt{np} \wedge 1 \le i_r \le \mathtt{mb}$.
Then we use $\mathsf{RelMax2}_{\ll}$ to compute $\max_{\ll}$ of two source functions (Appendix A.2). The result is

$$\begin{array}{l} S_2[p_r-1, q_r, i_r] \to S_1[p_r, q_r, i_r] \mid \\ \quad 2 \le p_r \le \mathtt{np} \wedge q_r = p_r-1 \wedge 1 \le i_r \le \mathtt{mb} \\ S_1[p_r-1, q_r, i_r] \to S_1[p_r, q_r, i_r] \mid \\ \quad p_r \le \mathtt{np} \wedge 1 \le q_r \le p_r-2 \wedge 1 \le i_r \le \mathtt{mb} \end{array} \tag{7}$$

$NotCovered = (p_r = q_r = 1 \wedge 1 \le i_r \le \mathtt{mb})$.

ω_5: $(C_0 \wedge \omega_5 \wedge NotCovered) = (q_w = q_r = p_r = 1 \wedge 1 \le i_w = i_r \le \mathtt{mb})$. This easily computes to dependence relation $S_0[1, i_r] \to S_1[1, 1, i_r] \mid 1 \le i_r \le \mathtt{mb}$. Finally $NotCovered$ = False.

After ω_5 step all the read instances of S_1 are covered and we don't have to compute dependences for ω_6 and any writes which textually precede S_0. The resulting source function for S_1 is given in (2).

5 Non-affine fragments

In this section we present our techniques for computing value-based dependences in non-affine program fragments.

What is a symbolic constant? Variable which is not assigned anywhere within the program fragment that we analyze is called *symbolic constant*. In the previous sections we held a traditional point of view that the analyzed fragment is the whole body of procedure or function. Now when we want to do a better job of dependence analysis for non-affine program fragments, we give a *dynamic* definition of analyzed fragment and symbolic constant.

The *unfixed zone* of depth d around statement S (denoted $UnFixed(S, d)$) is a loop nest which consists of statements belonging to d innermost loops surrounding S. The *fixed zone* of depth d around statement S (denoted $Fixed(S, d)$) consists of statements not belonging to $UnFixed(S, d)$. If $d = 0$ then unfixed zone is empty and everything around S is fixed.

A scalar variable v that is last written (defined) in the $Fixed(S, d)$ is considered to be a symbolic constant for the statement S at the depth d (denoted $v \in SymConst(S, d)$). To find the definition for the particular read of scalar variable and to distinguish between different definitions of the same variable we use the *Static Single Assignment (SSA)* graph of the program [Wol92].

This dynamic definition of symbolic constant is used in our dependence analysis algorithm in the following way. When computing the execution condition for the write statement in line 17 all the variables v such that $v \in SymConst(S, FixLoops)$ are considered to be symbolic constants.

Example: non-affine conditions. Let's consider program fragment in Figure 3(a). Computing the dependence relation for the statement S_3, we start with both loops `i` and `j` fixed: $i_w = i_r \wedge j_w = j_r$. Therefore unfixed zone of S_3 is empty and variable x is a symbolic constant. After visiting statements S_2 and S_1 we get dependences

$$\begin{array}{l} S_1[i, j] \to S_3[i, j] \mid 1 \le i, j \le n \wedge x \\ S_2[i, j] \to S_3[i, j] \mid 1 \le i, j \le n \wedge \neg x \end{array} \tag{8}$$

We discover that we do not have to unfix more loops because these 2 dependences cover all instances of read at S_3: $(1 \le i, j \le n) \wedge (x \vee \neg x) = (1 \le i, j \le n)$. Therefore we have proved that no loop-carried dependence exists from S_1 and S_2 to S_3.

However, dependences relation (8) is affine only if our scope is limited to the body of `j` loop. If we consider the whole program then dependence relation (8) becomes non-affine:

$$\begin{array}{l} S_1[i, j] \to S_3[i, j] \mid 1 \le i, j \le n \wedge x(i, j) \\ S_2[i, j] \to S_3[i, j] \mid 1 \le i, j \le n \wedge \neg x(i, j). \end{array} \tag{9}$$

This relation can not be represented within our framework which requires all constraints to be affine. Moreover, since we do not know which branch of IF statement S_0 is taken, we do not know exactly what instances of S_1 and S_2 are executed.

Computing the upper bound on iteration space. So we expand the actual iteration space to get rid of non-affine constraints, as it was suggested in [Voe92b]. For each non-affine expression in IF condition we assume that it can be both True and False, that is, we replace non-affine boolean terms with True in the positive context (that is, in the conjunction or disjunction), and with False in the negative context (that is, in the negation).

In the above example the upper bound on iteration space is $(S_1[i, j], S_2[i, j]) \mid 1 \le i, j \le n$, and the lower bound is empty.

Computing the lower and upper bounds on dependences. After we expanded the iteration space, we have to expand the set of dependences to make them affine too. That is, when dependence relation becomes non-affine as more loops are unfixed, we replace newly

Program	Lines	`f77 -O2`	[Fea91]	[PW93a]	Our algorithm	% of `f77 -O2` compile time	Times faster than [Fea91]	Times faster than [PW93a]
across	15	200	600	9	7.8	4	62	1.15
burg	29	600	5,600	91	82	14	56	1.10
relax	13	400	1,700	24	25	6	57	.96
gosser	22	700	2,800	62	50	8	43	1.24
choles	25	600	2,600	32	32	6	63	1.00
lanczos	69	1,700	12,600	119	115	7	88	1.03
jacobi	62	1,600	81,900	1,104	1,119	70	61	.98
btrix	155	8,600		1,515	1,290	15		1.17
cholsky	90	2,900		246	446	15		.55
vpenta	101	5,700		473	292	5		1.61
dctdx	72	1,440		324	145	10		2.23
ffa99	197	6,590		4,173	2,848	43		1.46
ffs99	192	6,460		4,817	3,419	53		1.40
interf	257	6,730		3,784	2,546	38		1.48
interf-hacked	279	6,790		4,092	2,538	37		1.61
ocean	12	820		25	23	3		1.09
olda	161	5,140		1,167	468	9		2.49
olda-hacked	154	4,800		796	723	15		1.10
poteng	194	10,820		1,670	1,293	12		1.29
poteng-hacked	212	11,040		1,952	1,535	14		1.27

Figure 8: Timing results (all times are in milliseconds)

non-affine variables with either True or False. Following [PW93b], we compute *lower* and *upper* bound for each dependence relation:

- Lower bound on dependence is computed by replacing non-affine variables with False in the positive context (in disjunctions and conjunctions) and with True in the negative context (in negations). That is, we over-constrain dependences to get lower bound.
- Upper bound is computed by replacing non-affine variables with True in positive context and with False in negative context. That is, we under-constrain dependences to get upper bound.

We use lower and upper bound on dependences in the following way:

- When we have to report non-affine dependence, we actually report *upper* bound on this dependence. So we add minimal number of dependences to make dependence relation affine.
- When computing what was covered by a write statement, we replace non-affine dependence with *lower* bound on it, because we can not be sure that any dependences between lower and upper bound really exist and cover read instances, and we know that dependences in the lower bound definitely exist.
- We do not compute $\max_{\ll}$ of the relations that contain affine approximations of constraints. It can not be done because we do not know exactly which statement instances described by these approximations are really executed. Instead we assume that all statement instances that are described by the approximated affine constraint are involved in the dependence. Doing so, we make dependence relation to bind many write statement instances to a read instance (instead of one). This is inevitable when affine approximations are used and this is the best we can do at the compile time.

For example the upper bound for the dependence relation (9) is: $\left[\begin{array}{l} S_1[i,j] \rightarrow S_3[i,j] \mid 1 \leq i,j \leq n \\ S_2[i,j] \rightarrow S_3[i,j] \mid 1 \leq i,j \leq n \end{array}\right.$. The lower bound for this relation is empty.

Example: non-affine subscripts. These techniques apply equally well to the non-affine IF conditions, loop bounds, and subscript functions.

Computing dependences for program in Figure 3(b) we start with loops `i` and `j` fixed and therefore we have a problem $1 \leq i_w = i_r, j_w = j_r \leq n \wedge x = x$, where x is a symbolic constant. Simplifying it we get affine dependence relation: $S_1[i,j] \rightarrow S_2[i,j] \mid 1 \leq i,j \leq n$. What's interesting, unfixing loops `i` and `j` does not make this dependence non-affine because x is not present in the resulting relation when loop `j` is unfixed. So non-affine fragments do not necessarily lead to inexact dependence relations.

6 How fast is our algorithm

We measured time taken by our dependence analysis algorithm to analyze Feautrier's benchmarks [Fea91] and some NASA NAS codes. In Figure 8 we compare our timing results with time taken by:

- Regular Fortran-77 compiler to compile the program [PW93a].
- Feautrier's algorithm to compute source functions for the program [Fea91].
- Pugh and Wonnacott techniques to compute memory-based direction vectors and value-based dependence relations for the program [PW93a].

Feautrier times were obtained on SUN Sparc ELC (SPECint89 rating of 18.0). All other measurements were performed on SUN SparcStation IPX (SPECint89 rating of 21.7).

7 Related work

We would like to compare our techniques to several other approaches to dependence analysis.

Memory-based dependence computation. Until recently only techniques for computing memory-based dependences were considered by most researchers [AK87, Wol82, MHL91]. The problem $SameCell(\mathbf{w}, \mathbf{r}, \mathbf{s})$ defined at line 17 of Figure 7 essentially describes a memory-based dependence. Since we compute this problem only once for each pair of statements, we don't take more time to compute memory-based dependences than existing techniques do.

In fact computing value-based dependences using our algorithm can take even less time than computing memory-based dependences when full cover is found quickly. Let's consider program in Figure 1(b). To compute dependence from S_{16} to S_{16} existing systems have to solve problem with 6 variables, and we know in advance that this dependence does not exist (as value-based) because statements S_1, ..., S_{14} cover S_{16} completely. So no time is spent disproving this dependence.

Feautrier work. Feautrier [Fea91] computes value-based dependences exactly for what we call affine program fragments, but his techniques are slow, because while computing dependences using definition (4), he does not keep track of what read instances were covered. So his algorithm always has to call PIP algorithm and analogue of $\mathsf{RelMax2}_{\ll}$ nd times, where n is number of candidate writes and d is number of loops surrounding read statement, while for us nd is upper bound which practically is never reached.

Also Feautrier's algorithm does not handle IF statements and non-affine program fragments.

Voevodins work. Voevodin & Voevodin [Voe92a, Voe92b] also compute exact value-based dependences for affine program fragments and they handle non-affine program fragments. They use methods that are close to that of Feautrier's. So we believe that our algorithm should work faster than theirs for the same reasons as above. Unfortunately, they do not describe their algorithm in detail and they do not provide timing results, so it is difficult to compare their algorithm to ours.

Maydan, Amarasinghe and Lam work. Their algorithm [MAL93, May92] does not apply to the general case of affine program fragment, so they use Feautrier's algorithm for backup. Their algorithm applies only to writes that do not self-interfere (that is, there is no output dependence from the write to itself) when unused loop indices are removed.

Our algorithm is also quick for such writes, because unused loop variables do not add constraints to the problem that we solve, and non-interfering writes usually lead to equating write loop variables to read loop variables which further simplifies the problem. Also their algorithm does not seem to handle non-affine program fragments.

Pugh and Wonnacott work. Pugh and Wonnacott use kill analysis to compute exact dependence information, that is, they first compute memory-based dependences and then kill or refine them by techniques originally described in [PW92] and [PW93b]. Since their kill analysis in the worst case considers all write–killer–read *triples*, while we in the worst case consider only all write–read *pairs*, the kill analysis can be expensive. So they have incorporated our idea of keeping track of the read instances that were already covered by another dependence under the name of "partial covers". They combine the partial cover computation with their traditional kill analysis as described in [PW93a].

However, their approach is different from ours because they do not use $\mathsf{RelMax}_{\ll}$ functions, instead they use the Presburger arithmetic (it can be described as our DNF package minus $\mathsf{RelMax}_{\ll}$ functions plus $\forall$ and $\exists$ quantifiers that can appear at any level of the formula) to perform the kill analysis equivalents of these operations. They also use memory-based dependences to perform some quick kills as it was suggested in the earlier paper [PW92], while we do not need them at all. However, if need be, we can compute the memory-based dependences with ease.

Both approaches are implemented in the Tiny tool, originally developed by Michael Wolfe and then considerably enhanced in the University of Maryland, College Park, so it is possible to compare the timing results (see Figure 8).

Handling non-affine constraints. In [Voe92b] and [PW93b] the authors describe their techniques for computing value-based dependences for non-affine program fragments.

Voevodin [Voe92b] suggests that the algorithm graph (that is, iteration space plus dependences) for non-affine fragment should be extended to become affine, but he does not describe how this is achieved.

In [PW93b] the authors propose to compute upper and lower bounds on dependences. However, their techniques can not prove that dependence from statement S_1 to S_2 is not carried by loop `i` in Figure 3(a).

A number of papers [AK87, HP91, LT88] suggest using symbolically enhanced versions of GCD test and Banerjee's inequalities. These techniques work only for memory-based dependence analysis and they are inexact even in this domain.

8 Source code availability

The implementation of our algorithms is integrated into the UMCP version of the *Tiny* tool for dependence analysis and program transformations. It is freely available by anonymous FTP from directory `pub/omega/lazy` on the machine `ftp.cs.umd.edu`.

9 Conclusion

In this paper we presented the algorithm which computes exact value-based (data-flow) dependences for affine program fragments and good affine approximations of value-based dependences for non-affine program fragments.

The basic idea of the algorithm — to start searching for candidate writes in lexicographically close proximity of a read statement for which dependence is being computed, and to expand search space only if non-covered read instances remain — makes it both efficient and capable of handling non-affine subscript functions, loop bounds and conditions without slowing down.

It also makes dependence analysis insensitive to the program size. That is, the time spent by our algorithm does not depend on number of statements that write the array in question or on number of loops that surround read. The algorithm time depends, however, on how many writes reach the read and on how complicated the dependence relation is — both these characteristics are not a function of program size.

10 Acknowledgements

My thanks go to everyone, who helped me to write this paper. Specials thanks to William Pugh and David Wonnacott from the University of Maryland, in discussions with whom this paper was born. I also would like to thank Valentine and Vladimir Voevodin from the Research Computing Center of Moscow State University, whose research was very inspirational for me.

References

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

[AL93] Saman P. Amarasinghe and Monica S. Lam. Communication optimization and code generation for distributed memory machines. In *ACM '93 Conf. on Programming Language Design and Implementation*, June 1993.

[B+89] M. Berry et al. The PERFECT Club benchmarks: Effective performance evaluation of supercomputers. *International Journal of Supercomputing Applications*, 3(3):5–40, March 1989.

[Blu92] William Joseph Blume. Success and limitations in automatic parallelization of the Perfect benchmarks^{TM} programs. Master's thesis, Dept. of Computer Science, U. of Illinois at Urbana-Champaign, 1992.

[EHLP91] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the automatic parallelization of 4 Perfect benchmark programs. In *Proc. of the 4th Workshop on Programming Languages and Compilers for Parallel Computing*, August 1991.

[Fea88a] P. Feautrier. Parametric integer programming. *Operationnelle/Operations Research*, 22(3):243–268, September 1988.

[Fea88b] Paul Feautrier. Array expansion. In *ACM Int. Conf. on Supercomputing, St Malo*, pages 429–441, 1988.

[Fea91] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1), February 1991.

[Fea92a] Paul Feautrier. Some efficient solutions to the affine scheduling problem, Part I, One-dimensional time. *Int. J. of Parallel Programming*, 21(5), Oct 1992.

[Fea92b] Paul Feautrier. Some efficient solutions to the affine scheduling problem, Part II, Multidimensional time. *Int. J. of Parallel Programming*, 21(6), Dec 1992.

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

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

[LT88] A. Lichnewsky and F. Thomasset. Introducing symbolic problem solving techniques in the dependence testing phases of a vectorizer. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.

[MAL93] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array data-flow analysis and its use in array privatization. In *ACM '93 Conf. on Principles of Programming Languages*, January 1993.

[May92] Dror Eliezer Maydan. *Accurate Analysis of Array References*. PhD thesis, Computer Systems Laboratory, Stanford U., September 1992.

[MHL91] D. E. Maydan, J. L. Hennessy, and M. S. Lam. Efficient and exact data dependence analysis. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 1–14, June 1991.

[Mur71] Y. Muraoka. *Parallelism Exposure and Exploitation in Programs.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, February 1971.

[Pug91] William Pugh. Uniform techniques for loop optimization. In *1991 International Conference on Supercomputing*, pages 341–352, Cologne, Germany, June 1991.

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

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

[PW93a] William Pugh and David Wonnacott. An evaluation of exact methods for analysis of value-based array data dependences. In *Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

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

[Voe92a] Valentin V. Voevodin. *Mathematical Foundations of Parallel Computing.* World Scientific Publishers, 1992. World Scientific Series in Computer Science, vol. 33.

[Voe92b] Vladimir V. Voevodin. Theory and practice of parallelism detection in sequential programs. *Programming and Computer Software (Programmirovaniye)*, 18(3), May 1992.

[Wol82] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, October 1982.

[Wol92] Michael Wolfe. Beyond induction variables. In *SIGPLAN Conference on Programming Language Design and Implementation*, San Francisco, California, June 1992.

A Appendix: computing lexicographical maximum

A.1 $\max_{\ll}$ of relation

In Figure 9 we present the algorithm to compute lexicographical maximum of relation which is not a function from read to write instances. The core of the algorithm is the function $\mathsf{ProblemMax}_{\ll}$ which finds $\max_{\ll}$ of a single conjunct. The result is a DNF that establishes relation between maximized variables $\mathbf{w}$ and input parameters $\mathbf{r}$.

The number of variables that we have to maximize is m (line 3). The problem of maximizing these variables is solved variable by variable. That is, we begin with maximizing lexicographically senior variable w_1. When maximum for it is established, it becomes input variable and variable w_2 is maximized, and so on. Let w_l denote the variable that is currently being maximized. To maximize w_l, we project out all lexicographically junior variables $w_{l+1}, \ldots, w_m$. Then in every resulting convex problem we examine constraints on w_l.

If w_l is fixed by equality constraint involving only input variables and w_l itself, then for every value of input parameters only one value of w_l is defined. Therefore this value is the maximal value of w_l (lines 11–12).

If variable w_l is not fixed by equalities, then we consider inequalities that provide upper bound for w_l (line 14). For every upper bound we generate a problem in which $\leq$ operator is replaced with $=$. This problem describes conditions under which this upper bound is reached, so we add the original problem to it and send it to the output list (line 18).

The upper bound on w_l expressed as $aw_l \leq F(\ldots)$ is converted in line 18 to the maximum for w_l which is $\lfloor F(\ldots)/a \rfloor$. Since we can not represent the integer division in our framework, we use wild-card variable α if $a \neq 1$: $aw_l + \alpha = F(\ldots) \wedge 0 \leq \alpha \leq a-1$.

Example of the algorithm work. Here we demonstrate how our algorithm computes the result of (5): $\max_{\ll}((i,j) \mid 0 \leq i \leq M \wedge 0 \leq j \leq N \wedge 2i{+}j = k)$.

Parameters of the algorithm are: $\mathbf{w} = (i,j)$, $m=2$, $\mathbf{r} = (M,N,k)$, $n=3$, $p = (0 \leq i \leq M \wedge 0 \leq j \leq N \wedge 2i{+}j = k)$. To get upper bounds on i we project away j and find two upper bounds on i:

$$\begin{cases} 0 \leq & i & \leq M \\ k-N \leq & 2i & \leq k \end{cases}$$

Replacing $i \leq M$ with $i = M$, simplifying and adding the original problem we get the problem that describes when upper bound $i \leq M$ is reached:

$$p_1 = (2M \leq k \leq 2M{+}N \wedge 0 \leq M \wedge i = M \wedge j = k{-}2M)$$

Then to find conditions under which another upper bound on i is a maximum, we replace inequality $2i \leq k$ with $2i{+}\alpha = k \wedge 0 \leq \alpha \leq 1$. Simplifying, we get:

$$p_2 = (k{-}1 \leq 2i \leq k \wedge k{-}N \leq 2i \wedge 0 \leq i \leq M \wedge j = k{-}2i)$$

So after the first iteration of the loop l (lines 5–26) we have the list *OutMax* that consists of two conjuncts (p_1 and p_2) that describe maximal values of i.

On the 2nd iteration of loop l we maximize variable j considering i as an input variable. Both in p_1 and p_2

Relation $\mathsf{RelMax1}_{\ll}(W[\mathbf{w},\mathbf{s}] \to R[\mathbf{r},\mathbf{s}] \mid p(\mathbf{w},\mathbf{r},\mathbf{s}))$ Begin
  Relation $MaxRel = \{\emptyset\}$
  For ($cp(\mathbf{w},\mathbf{r},\mathbf{s})$ in conjuncts of $p(\mathbf{w},\mathbf{r},\mathbf{s})$) do
    $MaxRel = MaxRel \cup \{W[\mathbf{w},\mathbf{s}] \to R[\mathbf{r},\mathbf{s}] \mid \mathsf{ProblemMax}_{\ll}(\mathbf{w} \mid cp(\mathbf{w},\mathbf{r},\mathbf{s}))\}$
  EndDo
  Return ($MaxRel$)

Dnf $\mathsf{ProblemMax}_{\ll}(\mathbf{w} \mid p(\mathbf{w},\mathbf{r}))$ Begin
  Result is $\forall \mathbf{v},\mathbf{r} : OutMax(\mathbf{v},\mathbf{r}) \Leftrightarrow \mathbf{v} = \max_{\ll}(\mathbf{w} \mid p(\mathbf{w},\mathbf{r}))$
3: Integer $m := |\mathbf{w}|,\ \ n := |\mathbf{r}|$
4: Dnf $InMax(\mathbf{w},\mathbf{r})$, $OutMax(\mathbf{w},\mathbf{r}) := p$
5: For $l := 1$ to m do
6:   $InMax(\mathbf{w},\mathbf{r}) := OutMax;\quad OutMax(\mathbf{w},\mathbf{r}) :=$ False
8:   For ($CInMax(\mathbf{w},\mathbf{r})$ in conjuncts of $InMax$) do
9:     Dnf $p_1(\mathbf{w}[1..l],\mathbf{r}) := \pi_{\mathbf{w}[1..l],\mathbf{r}}(CInMax)$
10:     For ($cp_1(\mathbf{w}[1..l],\mathbf{r})$ in conjuncts of p_1) do
11:       If (cp_1 contains equality involving w_l and not involving wild-card variables) then
12:         $OutMax := OutMax \ \vee\ (cp_1 \ \wedge\ CInMax)$
13:       Else
14:         Let's represent cp_1 as a conjunction of Nub upper bounds on w_l and everything else:

$$cp_1 = cp_{\mathtt{other}} \ \wedge\ \bigwedge_{i=1}^{Nub}(a_{i1}w_l \leq c_i + \sum_{j=2}^{m} a_{ij}w_j + \sum_{j=1}^{n} b_{ij}r_j) \text{ where } a_{i1} > 0$$

17:         For $i := 1$ to Nub do
18:           $OutMax := OutMax \ \vee\ (CInMax \ \wedge\ cp_{\mathtt{other}} \wedge$

$$a_{i1}w_l + \alpha_i = c_i + \sum_{j=2}^{m} a_{ij}w_j + \sum_{j=1}^{n} b_{ij}r_j \ \ \wedge\ \ 0 \leq \alpha_i \leq a_{i1} - 1)$$

21:         EndDo
22:         If ($Nub = 0$) then $OutMax := OutMax \ \vee\ w_l = \infty$
23:       EndIf
24:     EndDo
25:   EndDo
26: EndDo
  Return ($OutMax$)

Figure 9: Lexicographical maximum of parametrized set of vectors

the variable j is fixed by equality, so these conjuncts go directly to the resulting DNF. Finally we obtain the dependence relation (6).

A.2 $\max_{\ll}$ of two source functions

In Figure 10 we present the algorithm $\mathsf{RelMax2}_{\ll}$ to compute the lexicographical maximum of two source functions represented as dependence relations.

We consider every possible pair of conjuncts $sl_1 \in L_1$ and $sl_2 \in L_2$. If ranges of these conjuncts intersect then we call function $\mathsf{RelMaxVar}_{\ll}$ to compute the lexicographical maximum in the intersection area and add it to the resulting relation $MaxRel$. Then range of the intersection is subtracted from both relations and the process is repeated.

Finally one of the relations becomes empty or the relation ranges do not intersect anymore. We add what is left of relations to the result, because the relation that does not provide value for particular read variables value is always lexicographically less than the relations that provides the value.

We call the function $\mathsf{RelMaxVar}_{\ll}$ to compute the maximum of the two *simple* relations over the intersection of their ranges [2].

When computing maximum of the two simple relations we start with comparing lexicographically senior write variables $\mathbf{w}_1[1]$ and $\mathbf{w}_2[1]$. We build a conjunct pd that lets us know the sign of $\Delta = \mathbf{w}_1[1] - \mathbf{w}_2[1]$. In the line 25 we compute constraints on variables $\mathbf{r},\mathbf{s}$ under which $\Delta > 0$ and therefore $L_1 \gg L_2$, then in line 26 — constraints under which $\Delta < 0$ and $L_1 \ll L_2$.

Then if for some values of $\mathbf{r},\mathbf{s}$ we have $\Delta = 0$, we can not decide at this level which source function produces greater value (line 27). So we compare the vari-

[2] The domain of source function is equal to the range of the relation that represents this function.

Relation $\mathsf{RelMax2}_{\ll}$(Relation L_1, Relation L_2) Begin
Result is $\forall \mathbf{w}, \mathbf{r}, \mathbf{s} : (W[\mathbf{w}, \mathbf{s}] \to R[\mathbf{r}, \mathbf{s}]) \in \mathsf{RelMax2}_{\ll}(L_1, L_2) \Leftrightarrow$
$(W[\mathbf{w}, \mathbf{s}] \to R[\mathbf{r}, \mathbf{s}]) \in (L_1 /\neg\mathsf{range}(L_2) \cup L_2 /\neg\mathsf{range}(L_1)) \vee W[\mathbf{w}, \mathbf{s}] = \max_{\ll}(L_1^{-1}(\mathbf{r}, \mathbf{s}), L_2^{-1}(\mathbf{r}, \mathbf{s}))$
1: Relation $MaxRel := \{\emptyset\}$
2: For ($sl_1 = \{W_1[\mathbf{w}, \mathbf{s}] \to R[\mathbf{r}, \mathbf{s}] \mid p_1(\mathbf{w}, \mathbf{r}, \mathbf{s})\}$ in simple relations of L_1) do
3: For ($sl_2 = \{W_2[\mathbf{w}, \mathbf{s}] \to R[\mathbf{r}, \mathbf{s}] \mid p_2(\mathbf{w}, \mathbf{r}, \mathbf{s})\}$ in simple relations of L_2) do
4: Relation $Cmax = \mathsf{RelMaxVar}_{\ll}(sl_1,\ sl_2,\ 1,$ (number of loops surrounding both W_1 and W_2))
5: If ($Cmax \neq \{\emptyset\}$) then
6: $MaxRel := MaxRel \cup Cmax$
7: $L_1 := L_1 \cap \neg\mathsf{range}(Cmax)$; $L_2 := L_2 \cap \neg\mathsf{range}(Cmax)$
8: Start loop 2 from the beginning
9: EndIf
10: EndDo
11: EndDo
12: Return ($MaxRel \cup L_1 \cup L_2$)

Relation $\mathsf{RelMaxVar}_{\ll}$(
Simple relation $sl_1 = \{W_1[\mathbf{w}_1, \mathbf{s}] \to R[\mathbf{r}, \mathbf{s}] \mid C_1(\mathbf{w}_1, \mathbf{r}, \mathbf{s})\}$,
Simple relation $sl_2 = \{W_2[\mathbf{w}_2, \mathbf{s}] \to R[\mathbf{r}, \mathbf{s}] \mid C_2(\mathbf{w}_2, \mathbf{r}, \mathbf{s})\}$, int *level*, int *maxLevel*) Begin
(* Compare the variables $\mathbf{w}_1[level]$ and $\mathbf{w}_2[level]$ assuming $\mathbf{w}_1[1..level{-}1] = \mathbf{w}_2[1..level{-}1]$ *)
21: Relation $MaxRel := \{\emptyset\}$
22: If ($level > maxLevel$) Return (If $W_1 \gg W_2$ then sl_1 Else sl_2 EndIf)
23: Dnf $pd(\mathbf{r}, \mathbf{s}, \Delta w) := \pi_{\neg \mathbf{w}_1, \mathbf{w}_2}(C_1(\mathbf{w}_1, \mathbf{r}, \mathbf{s}) \wedge C_2(\mathbf{w}_2, \mathbf{r}, \mathbf{s}) \wedge \Delta w = \mathbf{w}_1[level] - \mathbf{w}_2[level])$
24: If ($pd =$ False) Return($\{\emptyset\}$)
25: $MaxRel := MaxRel \cup \{W_1[\mathbf{w}_1, \mathbf{s}] \to R[\mathbf{r}, \mathbf{s}] \mid C_1(\mathbf{w}_1, \mathbf{r}, \mathbf{s}) \wedge \pi_{\neg \Delta w}(pd \wedge \Delta w > 0)\}$
26: $MaxRel := MaxRel \cup \{W_2[\mathbf{w}_2, \mathbf{s}] \to R[\mathbf{r}, \mathbf{s}] \mid C_2(\mathbf{w}_2, \mathbf{r}, \mathbf{s}) \wedge \pi_{\neg \Delta w}(pd \wedge \Delta w < 0)\}$
27: $MaxRel := MaxRel \cup \mathsf{RelMaxVar}_{\ll}($
$\{W_1[\mathbf{w}_1, \mathbf{s}] \to R[\mathbf{r}, \mathbf{s}] \mid C_1(\mathbf{w}_1, \mathbf{r}, \mathbf{s}) \wedge \pi_{\neg \Delta w}(pd \wedge \Delta w = 0)\}$,
$\{W_2[\mathbf{w}_2, \mathbf{s}] \to R[\mathbf{r}, \mathbf{s}] \mid C_2(\mathbf{w}_2, \mathbf{r}, \mathbf{s}) \wedge \pi_{\neg \Delta w}(pd \wedge \Delta w = 0)\}$, $level + 1$, $maxLevel$)
30: Return ($MaxRel$)

Figure 10: Lexicographical maximum of two parametrized source functions

ables $\mathbf{w}_1[2]$ and $\mathbf{w}_2[2]$ by recursively calling the function $\mathsf{RelMaxVar}_{\ll}$. The level of the variable that we currently compare is stored in the variable *level*. Finally, if $\mathbf{w}_1[1..maxLevel] = \mathbf{w}_2[1..maxLevel]$, then the lexical ordering of the statements is used to decide which source function is lexicographically greater (line 22).

Example of the algorithm work. When computing source function (7) we call the function $\mathsf{RelMax2}_{\ll}$ with the following arguments:

$$L_1 = \{S_1[p_{w1}, q_{w1}, i_{w1}] \to S_1[p_r, q_r, i_r] \mid C_1 = (p_{w1} = p_r - 1 \wedge q_{w1} = q_r \wedge i_{w1} = i_r \wedge 1 \le q_r < p_r \le \mathtt{np} \wedge 1 \le i_r \le \mathtt{mb})\},$$

$$L_2 = \{S_2[p_{w2}, q_{w2}, i_{w2}] \to S_1[p_r, q_r, i_r] \mid C_2 = (p_{w2} = q_{w2} = q_r \wedge i_{w2} = i_r \wedge 1 \le q_r < p_r \le \mathtt{np} \wedge 1 \le i_r \le \mathtt{mb})\}$$

Since $\mathsf{range}(L_1) = \mathsf{range}(L_2)$, we execute only one call $\mathsf{RelMaxVar}_{\ll}(L_1, L_2, 1, 3)$.

In $\mathsf{RelMaxVar}_{\ll}$ we start with comparing write variables p_{w1} and p_{w2}. We form the conjunct

$$pd = (\Delta w = p_{w2} - p_{w1} \wedge C_1 \wedge C_2), \qquad (10)$$

project away all write variables ($\{p, q, i\}_{w\{1,2\}}$), and using the Omega test find that $\Delta w \le 0$.

Adding to (10) the inequality $\Delta w < 0$ and simplifying we find that L_1 is greater than L_2 if

$$1 \le q_r \le p_r - 2 \wedge p_r \le \mathtt{np} \wedge 1 \le i_r \le \mathtt{mb}. \qquad (11)$$

After this we add inequality $\Delta w = 0$ to (10). Simplifying, we get that $p_{w1} = p_{w2}$ if

$$q_r = p_r - 1 \wedge 2 \le p_r \le \mathtt{np} \wedge 1 \le i_r \le \mathtt{mb}. \qquad (12)$$

Executing the recursive call to the $\mathsf{RelMaxVar}_{\ll}$ we find that $q_{w1} = q_{w2}$ and therefore it's not clear yet which source function is greater. Going down one more level we get that $i_{w1} = i_{w2}$. Being still undecided, we go one more level down and find that there are no more loop variables to compare. The source function L_2 is then declared to be a maximum when (12) holds because $S_2 \gg S_1$.