TECHNICAL REPORT NO. 337

Rule-based Data Dependence Analysis

by

Larry Tenny

October 1991

COMPUTER SCIENCE DEPARTMENT

INDIANA UNIVERSITY

Bloomington, Indiana 47405-4101

Rule-based Data Dependence Analysis

Larry Tenny

October 30, 1991

Abstract

In this paper we outline a new approach to data dependence analysis. Our techique involves using rules to derive the data dependence graph from information readily obtainable from a parse tree and flow analysis of a program. The rules presented are written in the Rex-UPSL language. We present a complete example of the derivation of a data dependence graph.

1 Introduction

The semantics of most programming languages impose a strict order on the execution of statements in the language. Except for specific transfer of control statements (eg. `goto`), or conditional statements (eg. `if-then-else`), the order imposed by the language is the textual order of the statements in the program. Arbitrary changes to the textual order of statements in a program change the behavior of the program. Thus the semantics of the programming language demand a particular order of execution.

This order might seem to pose a rather serious problem for compilers that attempt to optimize code since many of these optimizations result in a reordering of statements [1]. Compilers that attempt to automatically vectorize or parallelize code might seem doomed from the start since vectorization and parallelization at best requires some statements to be executed concurrently, and at worst, executed in a random order.

Under certain conditions, however, this ordering can be relaxed. Some statements may be executed concurrently or in a random order without

program	$\longrightarrow$	procedure+
procedure	$\longrightarrow$	procedure-declaration declaration* statement+
statement	$\longrightarrow$	assignment | conditional | do-loop
assignment	$\longrightarrow$	variable = expr
conditional	$\longrightarrow$	**if** **(** logical-expr **)** **then** statement+ **endif**
do-loop	$\longrightarrow$	**do** variable = expr **,** expr [, expr] statement+ **enddo**

Figure 1: Abstract syntax for our language model.

changing the semantics of the resulting program. Determining which of the statements in a given program can be re-ordered is the subject of this paper.

Data dependence analysis plays a key role in code optimization and program restructuring. Some aspects of data dependence analysis are employed in almost every optimizing compiler, but it finds heavy use in compilers that attempt automatic code vectorization and parallelization.

The central goal of data dependence analysis is to determine, as much as computationally possible, the nature and extent of data interactions between statements in a program. Some aspects of these interactions are undecidable. Some are computationally expensive. Some are simply a matter of choosing the appropriate representation. Data dependence analysis provides the higher level abstractions and operations access to information about these interactions.

We use a restricted version of Fortran as our language model (see Figure 1). We use Fortran because it is the language of choice in high performance computing environments that benefit most from the restructuring processes.

The Fortran we use is restricted in the sense that our model retains some, but not all of the semantic qualities of Fortran. For simplicity we do not allow any of the methods for aliasing memory locations (like `equivalence` or `common` blocks), non-structured loops which contain more than one induction variable, and anything but simple linear functions in array indices. These restrictions may seem severe, but in practice, most Fortran programs fit into this model.

The ideas presented here are not tied to the language model we have chosen. Indeed, these techniques apply to virtually all imperative languages.

Much about data dependence analysis is not covered in this paper. In particular, we do not address interprocedural analysis [15], semantic analysis [12], nor symbolic subscript analysis [8].

2 Scalar Analysis

We call variables that denote arrays **vector** variables and all other variables **scalar** variables. The importance of this distinction will become clear shortly. For now, we assume that all variables are scalar. Vectors will be handled after we have developed some basic tools.

Aside from specific control constructs like `if`, `goto` and `subroutine` or `function` invocation, the semantics of our model, and indeed of most programming languages, require statements to be executed in textual order. This requirement arises directly from the often implicit requirement that the order of the side-effects to the state of the program be predictable. Consider the following statements.

S_1: `z = x + y`
S_2: `a = z + b`

If S_2 is executed before S_1, the value in the variable `a` would likely be different than if S_1 is executed first. It would seem then, that the textual order of statements in a program mandate a fixed execution order. However, consider:

S_3: `z = x + b`
S_4: `a = c + b`

Here the statements may be executed in either order without changing the final values of `a` or `z`. We call this property **commutativity**.

A fundamental problem for us is to determine when two statements are commutative. Commutativity implies that parallel execution of the statements is semantically valid, assuming of course that other factors such as

I/O are not involved. To see why this might be the case, consider a parallel processing model in which two or more statements are randomly distributed to two or more processors for unsynchronized execution. Due to the probabilistic nature of such a model, it would be impossible to guarantee a specific execution order while at the same time maintaining concurrent execution (ie. no synchronization). If the statements are commutative we would require no such guarantee.

Not only is it important for us to determine when two statements are commutative, but it is also important to determine when two statements access common variables. We call this property **factorability**. Notice that both S_3 and S_4 access but do not modify variable b. We say that S_3 and S_4 are factorable.

In some parallel architectures, factorability may degrade performance almost to the point of serialization. In other parallel architectures, most notably those with a large data cache or an abundance of registers, we might expect an increase in performance due to the lower memory access latency.

In this section we describe a framework for analyzing the data dependencies in a program. Along the way we will assume that information about control is readily obtainable from a traditional control dependence analysis. For an in-depth treatment of control analysis see [1] and [18]. We will take the liberty of referring to data dependence analysis as simply dependence analysis unless the reference is unclear.

We begin our discussion with a definition.

Definition 1 *Let P be a program in our model.*

$$S \in P \iff S \text{ is a statement in } P$$
$$VAR(S) = \{v : v \text{ is an occurrence of a variable in } S\}$$

VAR(S) is the set of all occurrences of variables in S, not simply the set of variables in S. Each occurrence of a variable in a statement is unique. We might denote a variable occurrence by its node identifier in the program's parse tree. We could introduce notation like v_i, where the subscript would be unique for each occurrence of variable v in statement S. In order to streamline matters as much as possible we introduce notation at the set level and urge the reader to keep this subtle distinction in mind when we deal at the variable level. Later we will denote an occurrence of variable v in statement S by v_i for $1 \leq i \leq n$ where n is the number of occurrences of v in S.

Definition 2 *Let* P *be a program and* $S \in P$ *a statement.*

$$
\begin{aligned}
USE(S) &= \{v : v \in VAR(S) \wedge v \text{ used in } S\} \\
DEF(S) &= \{v : v \in VAR(S) \wedge v \text{ defined in } S\} \\
USE(S)_n &= \{n : n \text{ is the name of } v \wedge v \in USE(S)\} \\
DEF(S)_n &= \{n : n \text{ is the name of } v \wedge v \in DEF(S)\}
\end{aligned}
$$

Here the sets *DEF(S)* and *USE(S)* contain all the occurrences of variables defined and of variables used in *S*, respectively. The sets $DEF(S)_n$ and $USE(S)_n$ contain just the names of the variables that occur in *DEF(S)* and *USE(S)*. $DEF(S)_n$ and $USE(S)_n$ are the name projections of the corresponding sets.

We have not given a precise meaning of the term 'defined'. For now, we will avoid a giving this term a precise meaning other than to suggest that it refers to variables modified in a statement. After some important notation and results are introduced, we will return to this issue.

We use the following lemma in our proof of the Commutativity Theorem. The lemma says that we have captured all the variables in *S* with the *USE* and *DEF* sets.

Lemma 1 *If P is a program and* $S \in P$ *a statement then,*

$$\mathrm{USE(S)_n} \cup \mathrm{DEF(S)_n} = \{n : n \text{ is the name of a variable in S}\}$$

Proof In our language model, as in most programming languages, all variables that occur in a statement occur either in a *USE* context or in a *DEF* context, or both. Hence, all variables in *S* are in either $USE(S)_n$ or $DEF(S)_n$. □

Now with the sets $USE(S)_n$ and $DEF(S)_n$ and Lemma 1, we are able to state the following sufficient condition for commutativity.

Theorem 1 (Commutativity) *Let* S_1, $S_2 \in P$ *be statements of program P. Then a sufficient condition for commutativity is:*

$$
\begin{aligned}
(1) \qquad & (\mathrm{DEF(S_1)_n} \cap \mathrm{USE(S_2)_n}) \;\cup \\
& (\mathrm{USE(S_1)_n} \cap \mathrm{DEF(S_2)_n}) \;\cup \\
& (\mathrm{DEF(S_1)_n} \cap \mathrm{DEF(S_2)_n}) = \emptyset
\end{aligned}
$$

Proof Our aim here is to show that if (1) holds, then the execution of S_1 followed by S_2 is semantically equivalent to the execution of S_2 followed by S_1. That is, the state of the variables mentioned in S_1 and S_2 are the same after the pair of statements have been executed in either order.

If (1) holds, then it follows that each of the clauses is $\emptyset$. We take each clause in turn.

First, $(DEF(S_1)_n \cap USE(S_2)_n) = \emptyset$ implies that no variable defined in S_1 is used in S_2. Hence, execution order will not affect members of $USE(S_2)$.

Likewise, $(USE(S_1)_n \cap DEF(S_2)_n) = \emptyset$ implies that no variable defined in S_2 is used in S_1 and again, execution order will not affect members of $USE(S_1)$.

Finally, $(DEF(S_1)_n \cap DEF(S_2)_n) = \emptyset$ implies that S_1 and S_2 do not define common variables, so the state of variables in $(DEF(S_1) \cup DEF(S_2))$ after the execution of the statements in either order is the same.

These cases account for all variables in the *USE* and *DEF* sets of each statement, and by Lemma 1 account for all variables in the statements. □

Notice that Theorem 1 does not provide us with a necessary condition for commutativity, only a sufficient one. In fact, it has been shown that the more general problem is undecidable [4].

The clauses in (1) present three different conditions that cause S_1 and S_2 to fail the sufficient condition for commutativity. If any of the clauses is not $\emptyset$, then the statements would not meet the sufficient condition. We define a relation for each clause.

Definition 3 *Let S_1, $S_2 \in P$ be statements of program P.*

$$\begin{aligned} S_1\, \delta^t\, S_2 &\iff DEF(S_1)_n \cap USE(S_2)_n \neq \emptyset \\ S_1\, \delta^a\, S_2 &\iff USE(S_1)_n \cap DEF(S_2)_n \neq \emptyset \\ S_1\, \delta^o\, S_2 &\iff DEF(S_1)_n \cap DEF(S_2)_n \neq \emptyset \end{aligned}$$

If two statements are in the relations δ^t, δ^a, or δ^o, there is said to exist a **true** depedence, **anti**-dependence, or **output** dependence, respectively, from the first statement to the second. There is an implied direction in the dependence. Expressions like, $S_i\, \delta^o\, S_j$ are read as: "There is an output dependence from S_i to S_j" or "S_j is output dependent on S_i."

We would like to define a relation for the one remaining combination of the sets *DEF* and *USE*, namely, $(USE(S_1)_n \cap USE(S_2)_n) \neq \emptyset$, even though

this clause does not appear in (1). This is the relation we need to express factorability.

Definition 4 *Let S_1, $S_2 \in P$ be statements of program P.*

$$S_1 \, \delta^i \, S_2 \iff USE(S_1)_n \cap USE(S_2)_n \neq \emptyset$$

Definition 5 *Let S_1, $S_2 \in P$ be statements of program P, then we say that S_1 and S_2 are factorable if and only if $S_1 \, \delta^i \, S_2$.*

When $S_1 \, \delta^i \, S_2$ holds we say that there is an **input** dependence between S_1 and S_2. There is not an implied direction associated with δ^i as there is with the other δ relations.

Notice that the δ^i relation does not suggest any restrictions on the semantic validity of executing S_1 and S_2 in either order. Rather, it might suggest some underlying benefits or costs in their parallel execution depending on the architectural model of the computer on which they are executed.

Among the four relations, there is an important distinction between the true dependence δ^t, and the two relations, δ^a and δ^o. The δ^a and δ^o relations arise because our model allows variables to be re-used. If our model prohibited such re-use, as is the case in functional languages, we could reduce our discussion of dependence relations to δ^t and δ^i. This also suggests dependencies that arise from δ^a or δ^o can be eliminated from programs by introducing new variables in places where variables are re-used.

The δ^i relation is, on the other hand, an altogether different sort of dependence. It does not participate in Theorem 1, but certainly how code for parallel machines is restructured. Like δ^t, δ^i is a more fundamental relation which cannot be eliminated by a simple transformation.

Theorem 1 along with the dependence relations are a fundamental result for our work. But in order to apply Theorem 1 we need to examine the *USE* and *DEF* sets more closely.

For any statement $S \in P$, we might have $|DEF(S)| > 1$ or $|USE(S)| > 1$. We cannot simply refer to the statements themselves as the discrete objects of a dependence. Although we do this when we speak of a dependence from one statement to another. The object of the dependence must involve the variable which causes the dependence. Likewise, we cannot refer to a particular variable as the discrete object of a dependence because the same variable may occur more than once in a statement possibly giving rise to

more than one dependence. Since the same variable my appear more than once in a statement, perhaps both in $DEF(S)$ and in $USE(S)$, we need to distinguish between occurrences of the same variable in a statement. Each of these occurrences, of course, could participate in a dependence relation.

Fortunately, our definition of $VAR(S)$ allows $USE(S)$ and $DEF(S)$ to have the property such that each occurrence of a variable is represented. The following definitions allow us to speak of individual instances of variables in a program.

Definition 6 *Let P be a program with only scalar variables and $S \in P$ a statement. Define,*

$$\begin{aligned}
v \equiv v' &\iff v, v' \in \left(\bigcup_{S \in P} VAR(S) \right) \wedge v \textit{ has same name and scope as } v' \\
\mathcal{D} &= \{(S,v) : v \in DEF(S) \wedge S \in P\} \\
\mathcal{U} &= \{(S,v) : v \in USE(S) \wedge S \in P\} \\
\mathcal{D}_S &= \{(s,v) : (s,v) \in \mathcal{D} \wedge s = S\} \\
\mathcal{U}_S &= \{(s,v) : (s,v) \in \mathcal{U} \wedge s = S\}
\end{aligned}$$

We say that (S, v) is an instance of a variable v *in statement* S.

The intent here is that $\mathcal{D}$ represents instances of variable definitions in P, while $\mathcal{U}$ represents instances of variable uses in P. The sets $\mathcal{D}_S$ and $\mathcal{U}_S$ represent the instances in a particular statement, S. It should be clear that for any program P and statement $S \in P$, $\mathcal{D}_S \subseteq \mathcal{D}$ and $\mathcal{U}_S \subseteq \mathcal{U}$.

We promised a more concrete definition of the term 'defined' in the remarks following Definition 2. We are now in a position to fulfill that promise. Consider the following statements:

S_1: `z = x + y`
S_2: `a = z + b`

Earlier we suggested that S_1 and S_2 are not commutative and Definition 2 along Theorem 1 provided us with the insight. Definition 3 allowed us to categorize the dependence as $S_1 \delta^t S_2$. A problem arises when another statement is introduced between S_1 and S_2.

```
S1: z = x + y
S': z = c + d
S2: a = z + b
```

Now does $S_1\, \delta^t\, S_2$ still hold? The answer is no. The definition of **z** in S_1 does not reach S_2, hence S_1 and S_2 are commutative. However, notice that $S'\, \delta^t\, S_2$ holds. S' and S_2 are not commutative. This does not present us with as much of an insurmountable problem as one might think. We simply need to refine the meaning of 'defined' to include only the **reaching** definitions of a variable.

2.1 Reaching Definitions

The problem of determining the reaching definitions is vital in determining the data dependence in a program. We briefly look a one algorithm to determine reaching definitions in order to make clear what a reaching definition is.

Definition 7 *Let $S \in P$ be a statement in program P.*

$$\begin{aligned}
GEN[S] &= \mathcal{D}_S \\
KILL[S] &= \bigcup_{S' \neq S} \{(S', v') : v \equiv v' \wedge v \in DEF(S)\} \\
IN[S] &= \bigcup_{S' \in P} \{(S', v) : \exists \textit{ a path from } S' \textit{ to } S \textit{ where } v \textit{ is not KILLed}\} \\
OUT[S] &= GEN[S] \cup (IN[S] - KILL[S])
\end{aligned}$$

These four sets denote the new definitions that are generated (GEN) by this statement, the definitions which mask previous definitions (KILL), the definitions which reach this statement (IN), and the definitions which survive this statement (OUT).

Figure 2 is an iterative algorithm from [1] which computes the reaching definitions, the set IN[S], for each statement. The algorithm starts by initializing the sets GEN[S], KILL[S], and OUT[S] to the set of variables defined in S, for each $S \in P$. Until no changes are made to OUT[S] for any statement S, IN[S] is assigned the union of its predecessor's OUT set. OUT[S] is assigned

GEN[S] plus the difference between the set of variables that reach S and the set of variables that are KILLed by S.

```
/* initialize KILL[S], GEN[S], IN[S], and PRED[S] */
 for S in P
     OUT[S] = KILL[S] = GEN[S] = the set of variables defined in S
     IN[S] = ∅
     PRED[S] = the set of predecessors of S
 end
/* iterate through P until no changes are made */
 change = TRUE
 while(change)
     change = FALSE
     for S in P
      IN[S] = ⋃_{S'∈PRED[S]} OUT[S']
      OLD = OUT[S]
      OUT[S] = GEN[S] ∪ (IN[S] - KILL[S])
      if OUT[S] != OLD then change = TRUE
     end
 end
```

Figure 2: Iterative algorithm to compute reaching definitions.

One particularly efficient way to implement this algorithm is to represent the sets with bit vectors. We number each instance $\mathcal{D}_S$ and record the presence or absence of the ith instance in the set with the ith bit. The resulting representation consumes far less space than a list-oriented representation and set operations in the algorithm become simple bitwise logical operations.

Along with reaching definitions, we need to consider uses that *live* until a particular statement. That is, we need also consider reaching uses: uses of a variable with no intervening definition until some statement $S \in P$. The algorithm for computing reaching uses is similar to the one for reaching definitions.

Now that reaching definitions and reaching uses have been introduced, we can modify our previous definition of the δ relations as follows.

Definition 8 *Let S_1, $S_2 \in P$ be statements of program P. Let REACHD(S) be the set of instances of variable definitions that reach statement S with no intervening definition. Let $REACHD(S)_n$ be the name projection of REACHD(S). Let REACHU(S) be the set of instances of variable uses that reach statement S with no intervening definition. Let $REACHU(S)_n$ be the name projection of REACHU(S).*

$$
\begin{aligned}
S_1\, \delta^t\, S_2 &\iff (DEF(S_1)_n \cap REACHD(S_2)_n) \cap USE(S_2)_n \neq \emptyset \\
S_1\, \delta^a\, S_2 &\iff (USE(S_1)_n \cap REACHU(S_2)_n) \cap DEF(S_2)_n \neq \emptyset \\
S_1\, \delta^o\, S_2 &\iff (DEF(S_1)_n \cap REACHD(S_2)_n) \cap DEF(S_2)_n \neq \emptyset
\end{aligned}
$$

This definition of the δ relations is more precise, yet somewhat more complex than our previous definition. Throughout the remaining part of this paper, whenever we refer to a δ relation, we implicitly refer to this definition.

3 Subscript Analysis

In the previous section we restricted our attention to scalar variables. Here we shift the focus to dependence analysis in the presence of vectors, especially in the context of loops. Vectors, often called subscripted variables or arrays, are used extensively in loops. Loops offer a great potential source for speedup in parallel systems[10, 2, 11]. However, vectors used in the context of loops complicate the analysis considerably [16, 17, 18].

In this section we introduce some additional dependence notation along with a few new concepts. We derive one well known dependence test, the GCD Test, that can be used to disprove the existence of a dependence. The general problem of proving the existence of a dependence based on arbitrary subscript expressions is undecidable.

Consider the following loop.

```
do i = 1,10
 S1: z(i) = 2 * x(i)
 S2: y(i) = z(i) + 1
enddo
```

Intuitively we see that $S_1 \; \delta^t \; S_2$, and indeed if we simply assumed that z was a scalar variable using the results of the previous section we would conclude that S_2 is dependent on S_1. However, consider following slightly different loop.

```
do i = 1,10
 S1: z(i*2) = 2 * x(i)
 S2: y(i) = z(i*2+1) + 1
enddo
```

Here we would again conclude $S_1 \; \delta^t \; S_2$, but we see that S_1 defines the even elements of z while S_2 uses the odd elements of z. Our scalar analysis fails because we are forced to treat the entire array z as a single variable.

What we need to look at is not simply which variables are used or defined, an approach that was sufficient for scalars, but which positions within the vectors are used or defined. Since these positions are referenced by subscript expressions, the fundamental problem is to determine when these subscript expressions refer to the same element in an array. Since the subscript expressions can be functions of values computed at run-time the general problem is of course, undecidable.

3.1 Definitions

In our language model as in most programming languages, loops may be nested to virtually any depth. In practice, nesting to several levels is quite common so our approach should be general enough to handle loop nesting to an arbitrary depth.

Definition 9 *Let $L_1, \ldots, L_n$ be loops nested to depth n in program P. We define an iteration vector for $L_1, \ldots, L_n$ to be $\mathbf{i} = (i_1, \ldots, i_n)$ where each i_j in $1 \leq j \leq n$ is the value of the induction variable of loop L_j for some iteration of the loop.*

We say that $\{\mathbf{i} : \mathbf{i}$ is a vector in $L_1, \ldots, L_n\}$ is the **iteration space** *of the loop $L_1, \ldots, L_n$.*

An iteration vector describes the state of each induction variable in any particular iteration of the loop. There is a different iteration vector for each possible value of each induction variable. For example, in the following loop we have $\{\mathbf{i}\} = 10 \times 20 \times 5$.

```
L1: do i=1,10
L2:  do j=1,20
L3:   do k=1,5
...
```

Loops need not be, and often are not, perfectly nested. We might have a nesting similar to the loop below. Here the length of the iteration vector for L_1 is 1, but the length of the iteration vector for L_1, L_2 and L_1, L_3 is 2.

```
L1: do i=1,25
L2:  do j=1,4
S1:   a(j,i) = c * b(i,j)
     enddo
     c = e(i)
L3:  do j=1,4
      a(j,i) = c * d(i,j)
     enddo
    enddo
```

The association of an iteration vector with a statement is written as $S(\mathbf{i})$ and denotes a particular instance of the statement in an iteration of a loop. For example, $S_1((1,3))$ is the statement instance: `a(3,1) = c * b(1,3)`.

As in the previous section, the central problem is to determine when two statements are in a δ relation. We say that two statements $S_1, S_2 \in P$ are dependent if and only if there exists some pair of iterations vectors $(\mathbf{i}, \mathbf{i}')$ such that $S_1(\mathbf{i})\, \delta^t\, S_2(\mathbf{i}')$ or $S_1(\mathbf{i})\, \delta^a\, S_2(\mathbf{i}')$ or $S_1(\mathbf{i})\, \delta^o\, S_2(\mathbf{i}')$ holds. In addition, we say that two statements are input dependent if and only if there exists some pair of iteration vectors $(\mathbf{i}, \mathbf{i}')$ such that $S_1(\mathbf{i})\, \delta^i\, S_2(\mathbf{i}')$ holds.

Definition 10 *Let $S_j, S_k \in P$ be two statements in loop $L_1, \ldots, L_n$ and let $S_j(\mathbf{i})$, $S_k(\mathbf{i}')$ be a pair of statement instances where $(\mathbf{i}, \mathbf{i}')$ is in the iteration space of the loop. Define,*

$$\begin{aligned}
\mu_j &= i'_j - i_j \\
\theta_j &= \begin{cases} > & \mu_j < 0 \\ = & \mu_j = 0 \\ < & \mu_j > 0 \end{cases} \\
\boldsymbol{\mu} &= (\mu_1, \ldots, \mu_m), \text{ where } m = \min(|\mathbf{i}|, |\mathbf{i}'|) \\
\boldsymbol{\theta} &= (\theta_1, \ldots, \theta_m), \text{ where } m = \min(|\mathbf{i}|, |\mathbf{i}'|)
\end{aligned}$$

We call $\boldsymbol{\mu}$ the **distance** *vector and $\boldsymbol{\theta}$ the* **direction** *vector between $\mathbf{i}$ and $\mathbf{i}'$.*

Definition 11 *Let $*$ refer to an arbitrary direction $\theta \in \{<, >, =\}$. We define the set of plausible direction vectors, $\psi(S_j, S_k)$, between two statements $S_j, S_k \in P$ where S_j occurs textually before S_k in P as,*

$$\psi(S_j, S_k) = \{=, \ldots\} \cup \{=, \ldots, <, *, \ldots\} \cup \{<, *, \ldots\}$$

We will use the direction vector in stating the GCD Test. The direction vector is useful in categorizing a dependence between two statements. If the distance vector between two statment instances $S_j(i)$ and $S_k(i')$ is non-zero and $S_j(i)\ \delta\ S_k(i')$, then we say that the dependence is **loop-carried**. We denote a loop-carried dependence by δ_c where c is the distance μ between the iteration vectors. If the distance vector is zero, we say that the dependence is **loop-independent** and denote this by δ_∞.

3.2 The GCD Test

Our present challenge is to determine if two statements are dependent given the possible set of iteration vectors and the subscript expressions contained in the array references of the statements. The following loop illustrates the problem.

```
do i=1,10
 S1: a(8*i-2) = d
 S2: c = a(2*i)
enddo
```

We would like to know if there exists iteration vectors $(\mathbf{i}, \mathbf{i}')$ such that $S_1(\mathbf{i})\,\delta\,S_2(\mathbf{i}')$. In other words, we would like to know it there is a solution[1] to the following dependence equation, $8x - 2y = 2$ where $1 \leq x, y \leq 10$.

Recall that we limit subscript expressions to simple linear functions. In general, we can write the dependence equation as,

$$a_0 + \sum_{1 \leq j \leq n} a_j i_j = b_0 + \sum_{1 \leq j \leq n} b_j i'_j$$

which we may write in a more familiar form as,

$$\sum_{1 \leq j \leq n} a_j i_j - \sum_{1 \leq j \leq n} b_j i'_j = (b_0 - a_0)$$

which is a linear diophantine equation.

If there is a solution within the bounds of our induction variables then we know that a dependence exists. On the other hand, if we can prove that no solution exists, then no dependence exists. In order to develop this idea further, we need the following results.

Lemma 2 *Let*

$$\sum_{1 \leq i \leq n} a_i x_i = c$$

be a linear diophantine equation and $g = \gcd(a_1, \ldots, a_n)$, *then*

$$\exists v_1, \ldots, v_n : \sum_{1 \leq i \leq n} a_i v_i = g$$

Proof Let $\mathrm{C} = \{\sum_{1 \leq i \leq n} a_i v_i : v_i \in \mathcal{Z}\}$ and $c' = \min\{c \in C \wedge c \neq 0\}$. First, we claim that c' divides each a_i, $1 \leq i \leq n$. Assume that c' does not divide

[1] Let $\mathbf{i} = (2)$ and $\mathbf{i}' = (7)$.

a_j for some j in $1 \leq j \leq n$, then

$$\begin{aligned} a_j &= c'q + r \\ r &= a_j - c'q \\ &= a_j - (\sum_{1 \leq i \leq n} a_i v_i) q \\ &= a_j(1 - v_j)q + \sum_{\substack{i \neq j \\ 1 \leq i \leq n}} -a_i v_i q \end{aligned}$$

Hence $r \in C$, but this contradicts the definition of c'. Since a_j was chosen arbitrarily, it follows that c' divides each a_i.

Next, we claim that $g = \gcd(a_1, \ldots, a_n) = c'$. If $g < c'$ then c' is a greater common divisor of $a_1, \ldots, a_n$ which contradicts the definition of g.

If $g > c'$ then g does not divide c'. But, from the previous claim, c' divides $a_1, \ldots, a_n$ so g must divide c'. Hence, $g = c'$ and

$$\exists v_1, \ldots, v_n : \sum_{1 \leq i \leq n} a_i v_i = \gcd(a_1, \ldots, a_n) = g$$

□

Theorem 2 *Let*

$$\sum_{1 \leq j \leq n} a_j x_j = c$$

be a linear diophantine equation, and $g = \gcd(a_1, \ldots, a_n)$. *The equation has a solution if and only if* g *divides* c.

Proof If g divides c then $gk = c$ and from Lemma 2 there exists $v_1, \ldots, v_n$ such that $\sum_{1 \leq i \leq n} a_i v_i = g$. Hence,

$$gk = c = k \sum_{1 \leq i \leq n} a_i v_i$$

and the solutions are $(v_1 c/g, \ldots, v_n c/g)$.

If the equation has a solution $(u_1, \ldots, u_n)$ and g divides each a_i $1 \leq i \leq n$, then clearly g divides c □

Corollary 1 *Let*

$$\sum_{1 \le j \le n} a_j x_j = c$$

be a linear diophantine dependence equation associated with $S_m, S_n \in P$. If the equation has no solution then $S_m \, \delta \, S_n$ does not hold.

Theorem 2 provides us with a simple way to prove that no solution exists, and by Corollary 1 that no dependence exists. But when $\gcd(a_1, \ldots, a_n)$ does divide c, the theorem does not tell us where the solution is. In particular, it does not tell us if the solution exists within the iteration space of the loop. This problem is rather fundamental when dealing with vectors. The iteration space of a loop is not always known at compile time because the loop bounds may depend on values computed at run time. Using only the corollary, we are forced to take the conservative approach and assume a dependence exists whenever the equation has a solution, even though the solution may not fall within the iteration space of the loop.

Although the following theorem does not tell us if a solution exists within the iteration space, the GCD Test can indicate if the dependence is loop-carried or loop-independent.

Theorem 3 (GCD Test) *Let $a_1, \ldots, a_m$ and $b_1, \ldots, b_n$ be the coefficients of a dependence equation, θ be a direction vector between two statement instances $S(\mathrm{i})$ and $S'(\mathrm{i}')$, and $g = \gcd(\{a_j - b_j : \theta_j = \text{`='}\}, \{a_j : \theta_j \neq \text{`='}\}, \{b_j : \theta_j \neq \text{`='}\})$.*

If $S \, \delta \, S'$, then g divides $(b_0 - a_0)$.

Proof We may write the dependence equation as,

$$(2) \quad \sum_{1 \le j \le n} \{(a_j - b_j) i_j : \theta_j = \text{`='}\} \; + \; \sum_{1 \le j \le n} \{(a_j i_j : \theta_j \neq \text{`='}\} \\ - \sum_{1 \le j \le n} \{(b_j i'_j : \theta_j \neq \text{`='}\} = b_0 - a_0$$

Since (2) is a linear diophantine dependence equation associated with S and S', if $S(\mathrm{i}) \, \delta \, S'(\mathrm{i}')$ then by corollary 1 g divides $(b_0 - a_0)$. □

To illustrate how Theorem 3 might be used to distinguish between loop-carried and loop-independent dependencies, consider the following loop.

```
L1: do i=1,10
L2:  do j=1,10
S1:   a(2*i+3*j+2) = d
S2:   c = a(5*i+9*j+4)
     enddo
    enddo
```

The dependence equation is $2 + 2x_1 + 3x_2 = 4 + 5y_1 + 9y_2$. Which by theorem 2 has a solution since $\gcd(5, 9, -2, -3) = 1$ which divides 2.

If we choose $\boldsymbol{\theta} = (=, =)$ we have $\gcd(5-2, 9-3) = \gcd(3, 6) = 3$ which does not divide 2. By theorem 3 there is no dependence between $S_1(\mathbf{i}), S_2(\mathbf{i})$ and if a dependence exists within the iteration space it is a loop-carried dependence.

Another dependence test called the **Separability Test** [16] is based on the explicit representation of the solution space of the dependence equation. The Separability Test can be applied only if the dependence equation has exactly one induction variable. However, the test yields both a necessary and sufficient condition for dependence.

Finally, the **Banerjee Test** [3] can be used to determine of a solution exists within the iteration space but can not be used to determine if the solution is integer or real. Since we require that the solutions be integer, the Banerjee Test can only be used to disprove a dependence.

4 The Data-Dependence Graph

A data dependence graph (DDG) [9] is a digraph $G = (V, E)$ where $V = \{S_i \mid S_i \in P\}$ and $E = \{(S_i, S_j) \mid S_i\ \delta\ S_j\}$. Edge direction in G corresponds to the implied direction of the dependence. We label each edge with the corresponding dependence relation: δ^t, δ^a, δ^o, or δ^i (see Figure 3).

The DDG serves as the abstract representation of the data-dependencies in the program. As such, the DDG represents the absence of commutativity and the presence of factorability among statements in a program. Three of the relations represented in the DDG namely, δ^t, δ^a, δ^o, characterize the cause of non-commutativity. The δ^i relation represents factorability. The problem of finding the correct sequence of program transformations that result in

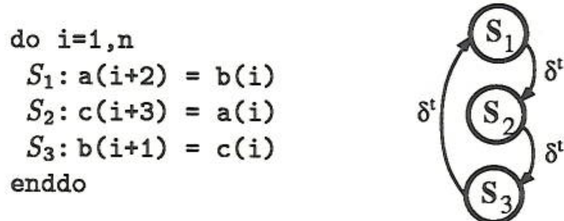

Figure 3: A data dependence graph.

some desired goal is essentially one of finding isomorphisms to a subgraph of the DDG. In this section we develop a concise set of rules for deriving the DDG from simple variable, data flow, and textual order information.

This approach is unique in many respects. On the practical side, it allows us to treat changes to the DDG in a uniform, well defined manner. This uniformity greatly simplifies the otherwise complex program dependence information and makes updating the dependences, especially after a series of program transformations, not only possible, but relatively easy.

A more fundamentally unique and advantageous feature of this approach is that it provides a clear, syntactic, and even executable definition of precisely what the DDG represents. The rules presented here to derive the DDG are analogous to the formal syntax rules presented when one describes a programming language.

Finally, an equally important advantage is that the representation presented here allows us to exploit the powerful pattern matching facilities inherent in rule-based languages to find the graph isomorphisms that are the essence of the restructuring process.

4.1 Preliminaries

The rules presented in this section are written in an experimental programming language developed specifically for this project. The language, called Rex-UPSL [14], supports both a traditional Scheme [13] environment as well as a tuple space similar to the tuple space of a relational database[5]. The tuple space is called **working memory** and denoted by $\mathcal{M}$. The individual tuples in $\mathcal{M}$, called **working memory elements**, are denoted by $\mathcal{M}_e$.

Definition 12 *A working memory element $\mathcal{M}_e$, is an ordered tuple of the form*

$$(class\ obj\ obj \ldots)$$

where class is a Scheme symbol and obj is a Scheme object.

Working memory, $\mathcal{M}$, is the collection of all working memory elements.

A particular rule in a Rex-UPSL program is applicable when the preconditions specified in the rule are satisfied by one or more tuples, $\mathcal{M}_e \in \mathcal{M}$.

Definition 13 *A rule is an expression of the following form.*

$$(\texttt{rule}\ name\ \underbrace{pattern\ pattern \ldots}_{LHS} \texttt{-->} \underbrace{expression}_{RHS})$$

The left hand side (LHS) is a conjunction of patterns and the right hand side (RHS) is an arbitrary Rex-UPSL expression.

Essentially, a rule consists of two parts: a left-hand side and a right-hand side. The left-hand side is a conjunction of patterns similar to a prototypical $\mathcal{M}_e$, but may contain variables or relational operators. The right-hand side is an expression that is a candidate for evaluation when the left-hand side patterns are consistently matched by some subset of $\mathcal{M}$.

It is important to note here that rules in Rex-UPSL are declarative objects in much the same sense as clauses in Prolog. Rules do not "check" $\mathcal{M}$ when first encountered by the Rex-UPSL interpreter. Rather, the set of satisfied rules is determined after each change to $\mathcal{M}$ using an efficient, state saving, many pattern, many object matching algorithm [7].

The syntax and semantics of our rule-based language will be illustrated here by way of example and discussion. A formal syntactic description is presented in the appendix. See [14] for a full discussion of the language.

4.2 Variables, Data Flow, and Textual Order

Programs contain variables. Edges in a DDG represent data dependencies that arise from the use and re-use of variables in a program. We begin our discussion by defining a representation for variables, data flow, and textual order in a program.

Definition 14 *Recall that a variable instance is the pair* (S_j, v_i) *where* $S_j \in P$ *and* v_i *is an occurrence of variable* v *in* S_j. *For each variable instance* (S_j, v_i), *let* $t_{(S_j,v_i)}$ *be a unique integer. We will use* $t_{(S_j,v_i)}$ *as a tag to refer to the instance* (S_j, v_i). *Likewise, let* t_v *and* t_{S_j} *be integers such that* $\forall_{v,v'}\ v \equiv v' \leftrightarrow t_v = t_{v'}$ *and* $\forall_{k \neq j}\ t_{S_j} \neq t_{S_k}$. *We will use* t_v *and* t_{S_j} *as tags to refer to variables and statements.*

Initially, $\mathcal{M}$ is empty. A parse of the program results in, among other things, a set of variable instances (S_j, v_i), the corresponding tags $t_{(S_j,v_i)}$, and a set of statement tags t_{S_j}. We define $\mathcal{M}_{e_u}$ as follows for variables used in P,

$$\mathcal{M}_{e_u} = \{(\texttt{var use}\ t_v\ t_{(S_j,v_i)}\ t_{S_j}) \mid S_j \in P \wedge v \in USE(S_j)\}$$

and $\mathcal{M}_{e_d}$ as follows for variables defined in P.

$$\mathcal{M}_{e_d} = \{(\texttt{var def}\ t_v\ t_{(S_j,v_i)}\ t_{S_j}) \mid S_j \in P \wedge v \in DEF(S_j)\}$$

Each $\mathcal{M}_e \in \mathcal{M}$ is a tuple with a class of `var`, a reference designator, `use` or `def`, a variable instance tag, and a statement tag. By Lemma 1, all variable instances in P are represented in $\mathcal{M}$ by $\mathcal{M}_{e_u} \cup \mathcal{M}_{e_d}$.

As discussed in Section 2, data flow is vital in determining data dependence and ultimately in building the DDG. Four types of data flow are important for our discussion. We represent a definition of a variable followed by a use without any intervening definition (see Figure 2) by the following working memory element,

$$\mathcal{M}_{e_f} = (\texttt{flow def-use}\ t_v\ t_{S_1}\ t_{S_2})$$

The class is `flow` and the remaining objects indicate the type of data flow (see Table 1), the variable, and the two statements associated with the flow.

With the program components represented in $\mathcal{M}$, most of the data dependencies in a program can be derived. However, without textual order information, we cannot distinguish between some loop-independent and loop-carried dependencies. Consider the following loop.

```
do i=1,100
 S1: a(i) = d
 S2: c = a(i)
enddo
```

Clearly $S_1 \; \delta^t \; S_2$ holds while $S_j \; \delta^a \; S_i$ does not. Before subscript analysis, data flow analysis finds both the definition of a in S_1 followed by the use of a in S_2 and, because of the loop, the use of a in S_2 followed by the definition of a in S_1. This would imply both $S_1 \; \delta^t \; S_2$ and $S_j \; \delta^a \; S_i$ hold. After subscript analysis we know that the dependence is loop-independent, but we still cannot conclude which of the dependencies hold.

The textual ordering of S_1 and S_2 is the missing link. If we can prove that S_1 textually occurs before S_2, we can establish $S_1 \; \delta^t \; S_2$.

We represent textual ordering in which S_2 follows S_1 with the following working memory element.

$$\mathcal{M}_{e_o} = (\texttt{order} \;\; t_{S_1} \;\; t_{S_2})$$

Notice that we have already established that there is a definition-free execution path from S_1 to S_2 and a similar path from S_2 to S_1.

Textual ordering is a weaker notion than execution ordering. Execution ordering is essential in determining the reaching definitions and reaching uses of a variable. The weaker textual order is sufficient to distinguish between loop-independent and loop-carried dependencies.

In the next section we show how the DDG can be derived from this representation of variable occurrences, flow information, and textual ordering (see Table 1). This is not the only representational set, nor is it the smallest. There are disadvantages in using this set. Most notably, we sacrifice the ability to handle variable aliases by not explicitly representing execution paths. The set we have chosen does allow us to use a much smaller collection of rules to derive the DDG. Essentially, we have pushed much of the complexity of the derivation into the builtin flow analysis at the price of reducing the number of things we can reason about.

4.3 Deriving the DDG

With a uniform representation in $\mathcal{M}$ for the variables, data flow, and textual ordering, in this section we introduce several rules for deriving the DDG. Deriving a particular dependence edge in the DDG is a two step process. First, a tentative dependence edge is built if and only if the preconditions for the edge are present in $\mathcal{M}$. Next, the set of tentative dependence edges become the preconditions for the formation of the final dependence edges.

$\mathcal{M}_e$	Description
(var use t_v $t_{(S,v_i)}$ t_S)	$v \in \text{USE}(S)$
(var def t_v $t_{(S,v_i)}$ t_S)	$v \in \text{DEF}(S)$
(flow def-use t_v t_{S_1} t_{S_2})	$v \in \text{DEF}(S_1) \wedge v \in \text{USE}(S_2)$
(flow def-def t_v t_{S_1} t_{S_2})	$v \in \text{DEF}(S_1) \wedge v \in \text{DEF}(S_2)$
(flow use-def t_v t_{S_1} t_{S_2})	$v \in \text{USE}(S_1) \wedge v \in \text{DEF}(S_2)$
(flow use-use t_v t_{S_1} t_{S_2})	$v \in \text{USE}(S_1) \wedge v \in \text{USE}(S_2)$
(order t_{S_i} t_{S_j})	S_j textually follows S_i

Table 1: Summary of $\mathcal{M}_e$ representations of variables, data flow, and control.

We begin with a rule that builds tentative true dependence edges. The preconditions for a tentative true dependence are as follows. Variable v is defined in some statement S_i and used in some statement S_j. It may or may not be the case that $i = j$. Also, there exists an execution path free of definitions to variable v from S_i to S_j.

In Rex-UPSL, these preconditions are expressed in the following rule.

Rule 1

```
(rule true
  (var def <v> <vt1> <st1>)
  (var use <v> <vt2> <st2>)
  (flow def-use <v> <st1> <st2>)
-->
  (make tentative-dependence true (characterize <vt1> <vt2> <st1> <st2>)
   <vt1> <vt2> <st1> <st2>))
```

First, a note on syntax. The patterns on the left-hand side that are the preconditions for the application of the rule, look like memory elements except for the brackets < > surrounding some of the identifiers. These brackets denote variables in the pattern. Variables mentioned more than once on the left-hand side must match the same value. Once matched consistently, these variables are available for use in the right-hand side expression.

Here the left-hand side has three patterns. The first and second match any pair of instances of a particular variable in which the variable occurs in

Characterization	Description
scalar	scalar variable
loop-independent	$\theta = (=, \ldots)$
positive	$\theta \in \{(=, \ldots, >, *)\} \cup \{(>, *, \ldots)\}$
negative	$\theta \in \{(=, \ldots, <, *)\} \cup \{(<, *, \ldots)\}$
non-linear	non-linear subscript expression
unknown	dependence unknown
no-dependence	no dependence

Table 2: Characterization of dependence directions.

a `def` context and a `use` context, respectively. The third pattern matches any of our `flow` memory elements involving the definition of the variable in one statement `<st1>`, followed by a use of the variable in another statement `<st2>`, with no intermediate definition.

Because multiple occurrences of a pattern variable must match the same value, the first two patterns match instances of the same variable. The integer tags representing the instances are bound to the pattern variables, `<vt1>` and `<vt2>`. Likewise, because the pattern variable `<v>` must be bound consistently, the third pattern restricts which memory elements the first and second patterns can match. These involve a definition of `<v>` in statement `<st1>` followed by a use of `<v>` in statement `<st2>` with no intervening definition along any execution path from `<st1>` to `<st2>`.

On the right hand side, the special form `make` builds a memory element from its arguments. The function `characterize` determines if the variable reference is scalar or vector. If vector, the GCD Test is applied to the corresponding dependence equation. If a dependence is found it is characterized based on the direction vector of the dependence (see Table 2).

The following rules for tentative anti, output, and input dependencies have a similar structure.

Rule 2

```
(rule anti
  (var use <v> <vt1> <st1>)
  (var def <v> <vt2> <st2>)
  (flow use-def <v> <st1> <st2>)
```

```
  -->
    (make tentative-dependence anti (characterize <vt1> <vt2> <st1> <st2>)
       <vt1> <vt2> <st1> <st2>))
```

Rule 3

```
  (rule output
    (var def <v> <vt1> <st1>)
    (var def <v> <vt2> <st2>)
    (flow def-def <v> <st1> <st2>)
  -->
    (make tentative-dependence output (characterize <vt1> <vt2> <st1> <st2>)
       <vt1> <vt2> <st1> <st2>))
```

Rule 4

```
  (rule input
    (var use <v> <vt1> <st1>)
    (var use <v> <vt2> <st2>)
    (flow use-use <v> <st1> <st2>)
  -->
    (make tentative-dependence input (characterize <vt1> <vt2> <st1> <st2>)
       <vt1> <vt2> <st1> <st2>))
```

The rules we have stated are rather general. A specific example of a rule match resulting in the creation of a tentative anti-dependence edge is as follows.

Suppose we have two statement nodes with tags t_{S_1} and t_{S_2} that reference a variable t_v. Statement t_{S_2} assigns to t_v some value (i.e. $v \in \text{DEF}(S_2)$), while t_{S_1} reads from t_v (i.e. $v \in \text{USE}(S_1)$). $\mathcal{M}$ would then contain the following three memory elements,

$$\mathcal{M}_{e_1} = (\texttt{var use } t_v \;\; t_{v_i} \;\; t_{S_1})$$

$$\mathcal{M}_{e_2} = (\texttt{var def } t_v \;\; t_{v_j} \;\; t_{S_2})$$

$$\mathcal{M}_{e_3} = (\texttt{flow use-def } t_v \;\; t_{S_1} \;\; t_{S_2})$$

The first two preconditions of the `anti` rule are satisfied by the two elements, $\mathcal{M}_{e_1}$ and $\mathcal{M}_{e_2}$. These represent the two instances of the variable t_v

$\mathcal{M}_e$	Description
(order t_{S_i} t_{S_j})	S_j follows S_i
(var t_v $t_{(S,v_i)}$ *exprs* t_S)	$v \in \text{DEF}(S) \cup v \in \text{USE}(S)$
(flow def-use t_v t_{S_1} t_{S_2})	$v \in \text{DEF}(S_1) \wedge v \in \text{USE}(S_2)$
(flow def-def t_v t_{S_1} t_{S_2})	$v \in \text{DEF}(S_1) \wedge v \in \text{DEF}(S_2)$
(flow use-def t_v t_{S_1} t_{S_2})	$v \in \text{USE}(S_1) \wedge v \in \text{DEF}(S_2)$
(flow use-use t_v t_{S_1} t_{S_2})	$v \in \text{USE}(S_1) \wedge v \in \text{USE}(S_2)$
(dependence true t_{S_1} t_{S_2} t_{v_i} t_{v_j})	$S_1\ \delta^t\ S_2$
(dependence anti t_{S_1} t_{S_2} t_{v_i} t_{v_j})	$S_1\ \delta^a\ S_2$
(dependence output t_{S_1} t_{S_2} t_{v_i} t_{v_j})	$S_1\ \delta^o\ S_2$
(dependence input t_{S_1} t_{S_2} t_{v_i} t_{v_j})	$S_1\ \delta^i\ S_2$
(dependence* true t_{S_1} t_{S_2} t_{v_i} t_{v_j})	$S_1\ \delta^t\ S_2$ (conditional)
(dependence* anti t_{S_1} t_{S_2} t_{v_i} t_{v_j})	$S_1\ \delta^a\ S_2$ (conditional)
(dependence* output t_{S_1} t_{S_2} t_{v_i} t_{v_j})	$S_1\ \delta^o\ S_2$ (conditional)
(dependence* input t_{S_1} t_{S_2} t_{v_i} t_{v_j})	$S_1\ \delta^i\ S_2$ (conditional)

Table 3: Summary of dependence-related $\mathcal{M}_e$ types.

that occur in statements t_{S_1} and t_{S_2}. The pattern variables `<vt1>` and `<vt2>` are bound to the tags for use instance and the defining instance, respectively.

The third precondition is satisfied by $\mathcal{M}_{e_3}$. This element represents a reaching use of t_v between statements t_{S_1} and t_{S_2}.

Symbolically, the rule satisfaction is,

$$\mathcal{M}_{e_1} \wedge \mathcal{M}_{e_2} \wedge \mathcal{M}_{e_3} \longrightarrow \texttt{tentative-dependence anti...}$$

Input dependencies do not inhibit parallelization or vectorization, but their representation is important in the analysis of secondary memory effects.

The `characterize` procedure applies the GCD Test to subscript expressions to characterize the dependence. Constant subscript expressions or the absence of subscript expressions altogether imply scalar reference. But when subscript expressions are involved, the GCD Test along with some simple heuristics can, in many cases, determine the direction vector of a dependence if a dependence exists. This characterization of the dependence is used to determine the veracity of tentative dependence edges produced by the rules presented above. To see why this is important, consider the following loop.

```
do i=1,100
 S1 :a(i) = 3
 S2 :b = a(i-1)
enddo
```

Data flow analysis provides us with the following two `flow` working memory elements.

$$(\texttt{flow def-use}\ t_a\ t_{S_1}\ t_{S_2})$$

$$(\texttt{flow use-def}\ t_a\ t_{S_2}\ t_{S_1})$$

And in the context of the previous rules, the following tentative dependence edges are added to $\mathcal{M}$.

$$(\texttt{tentative-dependence true negative}\ t_{a_i}\ t_{a_j}\ t_{S_1}\ t_{S_2})$$

$$(\texttt{tentative-dependence anti positive}\ t_{a_i}\ t_{a_j}\ t_{S_2}\ t_{S_1})$$

Both tentative dependencies cannot hold. We know from Definition 11 that dependence direction vectors of the form $\{=,\ldots,>,\ldots\}$ or $\{>,*,\ldots\}$ are not in $\psi(S_1,S_2)$ hence are not plausible. We conclude that the anti-dependence does not exist. The following rule captures this notion.

Rule 5

```
(rule loop-carried
   (tentative-dependence <type> negative <vt1> <vt2> <st1> <st2>)
 -->
   (make dependence <type>  <vt1> <vt2> <s1> <st2>))
```

For tentative dependencies that are scalar or loop-independent, we need not consider the plausibility of the direction vector since the dependence is among statements in the same iteration of the loop. However, we do need consider the textual order of the statements for two reasons.

We need to remove the tentative self-dependencies. Self-dependencies arise from data flow analysis entering the `flow` elements into $\mathcal{M}$ for multiple references of a variable within a statment. Such references are significant

if they result in loop-carried dependencies, but not if they would result in loop-independent dependencies.

As mentioned above, the textual order of two statements involved in a tentative dependence is crucial in choosing between two possible dependence types. In $\mathcal{M}$ the execution order of each pair of neighboring statements is represented by,

$$\mathcal{M}_{e_o} = (\texttt{order}\ \ t_{S_1}\ \ t_{S_2})$$

Just textual neighbors are represented and not all possible statement pairs. Notice that the order relation is transitive. That is,

$$(\texttt{order}\ \ S_1\ \ S_2) \wedge (\texttt{order}\ \ S_2\ \ S_3) \rightarrow\ S_3 \text{ follows } S_1$$

To prove an textual ordering between two arbitrary statements S_i and S_j, we form the transitive closure over the order relation from S_i to S_j.

Rule 6

```
(rule transitive-closure
  (prove-order <st1> <st2>)
  (order <st1> <st>)
  (order <st> <st3> <> <st2>)
  -(order <st1> <st3>)
-->
  (make order <st1> <st3>))
```

Rule 6 has four preconditions. First, the current subgoal must be to prove an ordering between two statements, S_i and S_j. Next, there must exist an ordering between S_i and S_m and between S_m and S_n where $n \neq j$. This last condition precludes rule satisfaction when the desired order relation already exists. This last precondition requires that the derived order relation does not already exist in $\mathcal{M}$. Syntactically, this negated pattern is signaled by the proceeding -.

For the transitive-closure rule to become active, $\mathcal{M}$ must contain the subgoal prove-order. The first rule of the following rule pair establishes this subgoal when a tentative loop-independent dependence arises.

Rule 7

```
(rule possible-loop-independent
   (tentative-dependence <type> loop-independent <vt1> <vt2> <st1> <st2>)
 -->
   (make prove-order <st1> <st2>))
```

If an execution ordering between two statements involved in a tentative loop-independent dependence has been established, the following rule adds the appropriate dependence edge to the DDG.

Rule 8

```
(rule loop-independent
   (tentative-dependence <type> loop-independent <vt1> <vt2> <st1> <st2>)
   (order <st1> <st2>)
 -->
   (make dependence <type> <vt1> <vt2> <st1> <st2>))
```

For scalar variables we need not consider execution order because both tentative dependencies hold.

Rule 9

```
(rule scalar
   (tentative-dependence <type> scalar <vt1> <vt2> <st1> <st2>)
 -->
   (make dependence <type> <vt1> <vt2> <st1> <st2>))
```

As discussed above, the essential problem in determining a dependence between array references is finding the solution to the linear diophantine equation corresponding to the subscript expressions used in the array references. Often, however, it is not possible to determine if the dependence equation has a solution within the iteration space of a loop. In order to guarantee that the program transformations based on the DDG preserve the semantics of the program, we are forced to error on the conservative side and create a dependence edge.

This gives rise to two distinct kinds of edges in our dependence graph. One edge represents a proven dependence, that is, we can prove that the dependence equation has a solution and that a solution lies within the iteration space. The second type of edge represents a suspected, yet unproven

dependence. We can show that the dependence equation has a solution, but cannot prove or disprove a solution within the iteration space. Traditional data dependence analysis does not distinguish between these two types of edges.

If the GCD Test is unable to prove or disprove a dependence either because the subscript expression is non-linear or because the loop bounds are not known at compile time, the dependence is characterized as either `unknown` or `non-linear`. The following rule creates the conditional dependence.

Rule 10

```
(rule conditional
   (tentative-dependence <type> << unknown non-linear >>
      <vt1> <vt2> <st1> <st2>)
 -->
   (make dependence* <type> <vt1> <vt2> <st1> <st2>))
```

Note the use of the or predicate `<< >>` in the left hand side. The pattern matches any tentative dependence edge characterized as `unknown` or `non-linear`.

4.4 An Example

In this section we present a brief sketch of the DDG derivation for the following loop.

```
do i=1,100
 S1: a(i) = a(i+1)
 S2: b(2*i) = a(i)
 S3: c = b(2*i+1)
 S4: a(i+1) = b(2*i)+c
enddo
```

We assume that a parse tree has been constructed and the necessary information such as statement, variable, and instance tags, is readily available. Further, we assume that the variable and data analysis has been performed

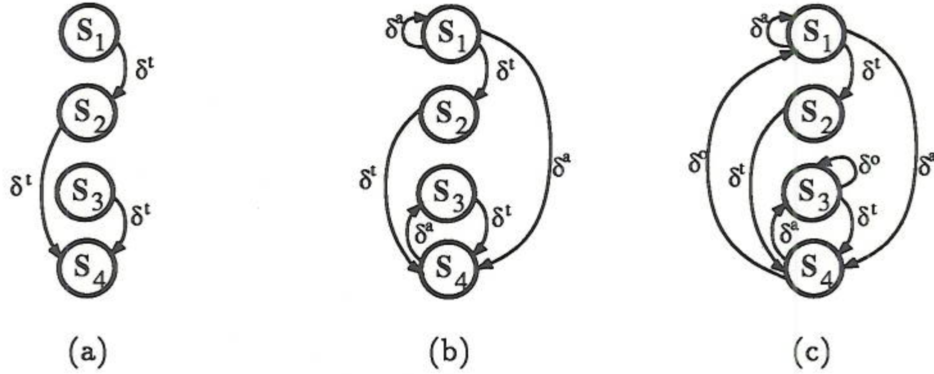

Figure 4: Derivation of the Data Dependence Graph.

and the appropriate `var`, `flow`, and `order` elements are in $\mathcal{M}$. In what follows, we present the derivation of each type of dependence edge separately. For brevity we do not present the derivation of input dependence edges.

We begin with the true dependence edges (Figure 4a). The definition of `a(i)` in S_1 and the use of `a(i)` in S_2 along with the assertion from flow analysis that the definition in S_1 of `a` reaches the use of `a` in S_2 satisfies the preconditions of Rule 1.

The `characterize` function performs the subscript analysis and concludes that if a dependence exists, it is loop-independent. The following `tentative-dependence` element is added to $\mathcal{M}$.

$$\mathcal{M}_{e_1} = (\texttt{tentative-dependence true loop-independent } t_{a_i}\ t_{a_j}\ t_{S_1}\ t_{S_2})$$

In a similar manner, the preconditions for tentative dependencies from S_2 to S_4, from S_3 to S_4, and from S_2 to S_3 are satisfied resulting in,

$$\mathcal{M}_{e_2} = (\texttt{tentative-dependence true loop-independent } t_{b_i}\ t_{b_j}\ t_{S_2}\ t_{S_4})$$

and,

$$\mathcal{M}_{e_3} = (\texttt{tentative-dependence true scalar } t_{c_i}\ t_{c_j}\ t_{S_3}\ t_{S_4})$$

and,

$$\mathcal{M}_{e_4} = (\texttt{tentative-dependence true no-dependence } t_{b_i}\ t_{b_j}\ t_{S_2}\ t_{S_3})$$

respectively.

Both $\mathcal{M}_{e_1}$ and $\mathcal{M}_{e_2}$ satisfy the preconditions for Rule 7 and the subgoals to prove that S_2 follows S_1 and S_4 follows S_2 in a textual ordering are entered into $\mathcal{M}$. In both cases, Rule 6 finds the ordering and enters the appropriate `order` elements into $\mathcal{M}$.

Finally, the preconditions for Rule 8 are satisfied and the true dependence edges are added to the DDG.

Determining the veracity of $\mathcal{M}_{e_3}$ is not so arduous. Rule 9 does not require the ordering assertion. It simply adds the true dependence edge to the DDG.

For $\mathcal{M}_{e_4}$, subscript analysis was sufficient to determine that no dependence exists.

The anti-dependence edges (Figure 4b) are derived in a similar way. Rule 2 enters the following elements into $\mathcal{M}$.

$$\mathcal{M}_{e_5} = (\texttt{tentative-dependence anti negative } t_{a_i}\ t_{a_j}\ t_{S_1}\ t_{S_1})$$

$$\mathcal{M}_{e_6} = (\texttt{tentative-dependence anti scalar } t_{c_i}\ t_{c_j}\ t_{S_4}\ t_{S_3})$$

$$\mathcal{M}_{e_7} = (\texttt{tentative-dependence anti loop-independent } t_{a_i}\ t_{a_j}\ t_{S_1}\ t_{S_4})$$

$$\mathcal{M}_{e_8} = (\texttt{tentative-dependence anti loop-independent } t_{a_i}\ t_{a_j}\ t_{S_2}\ t_{S_1})$$

Rule 5 is satisfied by $\mathcal{M}_{e_5}$ because subscript analysis characterized the direction of the dependence as negative, hence plausible for a loop-carried dependence.

Rule 7 along with Rule 6 attempt to establish an that S_4 follows S_1 and that S_1 follows S_2. Only the first of these succeeds.

The two output dependence edges (Figure 4c) are derived by the satisfaction of Rule 3 and Rule 5 for the loop-carried edge, and by Rules 3 and Rule 9 for the self-dependence edge.

5 Summary

In this paper we discussed scalar and vector data dependence analysis and derived one well known dependence test, the GCD Test. We described a rule-based approach to deriving a data dependence graph and presented a set of rules that formally describe the meaning of dependence through the

use of pre- and post-conditions. During our discussion, we illustrated the syntax and semantics of the experimental rule-based language Rex-UPSL.

Appendix: Rex-UPSL Syntax

Rex-UPSL is a programming language primarily designed for building large systems of rules, usually called production systems. The left-hand side patterns for rules in Rex are written using a syntax very similar to OPS5 [6]. The right-hand side actions of rules are written in Scheme [13]. In this appendix we present an extended Backus-Naur Form (BNF) syntax for Rex. The syntax for Scheme expressions is adapted from the offical Scheme report [13] and is integrated into the sytactic description of Rex-UPSL where appropriate.

An in depth discussion of the language and its uses can be found in [14].

The BNF presented here is extended with the following notation.

- Terminals are written in **boldface type**
- Descriptive terminals are written in *italics.*
- Items followed by + occur one or more times.
- Items followed by * occur zero or more times.

Syntax

program	⟶	statement*
statement	⟶	rule ∣ expression ∣ declaration
declaration	⟶	(**literal** lit-dcl+)
		∣ (**literalize** symbol symbol+)
		∣ (**vector-attribute** symbol+)
system	⟶	(**ps** statement+) ∣ (**named-ps** symbol statement+)
lit-dcl	⟶	symbol = number
tl-ce	⟶	(symbol tl-lhs-term*)
tl-lhs-term	⟶	^ symbol tl-lhs-value ∣ ^ number tl-lhs-value ∣ tl-lhs-value
tl-lhs-value	⟶	symbol ∣ num
rule	⟶	(**rule** symbol lhs **-->** rhs)
lhs	⟶	positive-ce ce*
ce	⟶	positive-ce ∣ negative-ce
positive-ce	⟶	form ∣ { element-var form } ∣ { form element-var }
negative-ce	⟶	- form

```
form                  →  ( symbol lhs-term* )
lhs-term              →  ^ symbol lhs-value | ^ number lhs-value | lhs-value
lhs-value             →  { restriction*} | restriction
restriction           →  << any-atom* >> | predicate atomic-value | atomic-value
atomic-value          →  ' symbol | var-or-const
var-or-const          →  symbol | number | variable
predicate             →  <> | = | < | <= | >= | > | <=>
rhs                   →  expression
token                 →  identifier | boolean | number | character | string
                         | ( | ) | #( | ' | .
deliminator           →  whitespace | ( | ) | " | ;
whitespace            →  space, newline, or tab
comment               →  ; characters up to newline
atomsphere            →  whitespace | comment
intertoken-space      →  atomsphere*
identifier            →  initial subsequent* | perculiar-identifier
initial               →  letter | special-initial
letter                →  a | b | ... | Z
special-initial       →  ! | $ | % | & | * | / | : | < | = > | ? | ~ | _ | ^
subsequent            →  initial | digit | special-subsequent
digit                 →  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
special-subsequent    →  . | + | -
expresison-keyword    →  quote | lambda | if | set! | begin | cond
                         | and | or | let | let* | letrec
                         | literalize | vector-attribute
variable              →  any identifier that is not a keyword
boolean               →  # t | # f
character             →  #\any-character
any-character         →  any character
number                →  + unumber | - unumber
unumber               →  digit digit*| digit . digit*
string                →  " any-character*"
symbol                →  identifier
datum                 →  simple-datum | compound-datum
simple-datum          →  boolean | number | character | string | symbol
compound-datum        →  list | vector
```

list	⟶	(datum*) | (datum datum*. datum) | ' datum
vector	⟶	#(datum*)
expression	⟶	variable | literal | procedure-call | lambda-expr
		| conditional | assignment | derived-expr | system
literal	⟶	quotation | self-evaluating
self-evaluating	⟶	boolean | number | character | string
quotation	⟶	' datum | (**quote** datam)
procedure-call	⟶	(operator operand*)
operator	⟶	expression
operand	⟶	expression
lambda-expr	⟶	(**lambda** formals expression*)
formals	⟶	(variable*) | variable | (variable variable*. variable)
conditional	⟶	(**if** expression expression expression)
assignment	⟶	(**set!** variable expression)
derived-expression	⟶	(**cond** cond-clause+)
		| (**cond** cond-clause*(**else expression+**))
		| (**and** expression*)
		| (**or** expression*)
		| (**let** (binding-spec) expression*)
		| (**let*** (binding-spec) expression*)
		| (**letrec** (binding-spec) expression*)
		| (**begin** expression*)
cond-clause	⟶	(expression*)
binding-spec	⟶	(variable expression)

References

[1] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers Principles, Techniques, and Tools.* Addison Wesley, 1985.

[2] ANDSHYH CHING CHEN, U. B., KUCK, D., AND TOWEL, R. Tiume and parallel processor bounds for fortran-like loops. *IEEE Trans. Comput. C-28*, 9 (1979).

[3] BANERJEE, U. Data dependence in ordinary programs. Master's thesis, University of Illinios at Urbana-Champaign, 1976.

[4] BERNSTEIN, A. Analysis of programs for parallel processing. *IEEE Trans. Electronic Computers 15* (1966), 757–62.

[5] DATE, C. *An introduction to database systems.* Addison-Wesley, 1983.

[6] FORGY, C. Ops5 user's manual. Tech. Rep. CMU-CS-81-135, Department of Computer Science, Carnegie-Mellon University, july 1981.

[7] FORGY, C. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence 19* (1982), 17–37.

[8] HAGHIGHAT, M. R. Symbolic dependence analysis for high performance parallelizing compilers. Tech. Rep. 995, CSRD, May 1990.

[9] KUCK, D., KUHN, R., PADUA, D., LEASURE, B., AND WOLFE, M. Dependence graphs and compiler optimizations. In *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages* (1981), pp. 207–218.

[10] KUCK, D., MURAOKA, Y., AND CHEN, S.-C. On the number of operations simultaneously executable in fortran-like programs and their resulting speedup. *IEEE Trans. Comput. C-21* (1972), 1293–1310.

[11] KUCK, D. J., SAMEB, A. H., CYTRON, R., VEIDENBAUM, A. V., POLYCHRONOPOULOS, C. D., LEE, G., MCDANIEL, T., LEASURE, B. R., BECKMAN, C., DAVIES, J. R. B., AND KRUSKAL, C. P. The effects of program restructuring, algorithm change, and architecture choice on program performance. In *1984 International Conference on Parallel Processing* (1984), pp. 129–138.

[12] PADUA, D. A., AND WOLFE, M. J. Advanced compiler optimizations for supercomputers. *Communications of the ACM 29*, 12 (December 1986).

[13] REES, J., AND CLINGER, W. Revised[3] report on the algorithmic language scheme. Tech. Rep. 174, Department of Computer Science, Indiana University, December 1986.

[14] TENNY, L. Upsl user's manual. Tech. Rep. 257, Department of Computer Science, Indiana University, Bloomington, Indiana, 1988.

[15] TRIOLET, R. Interprocedural analysis for program restructuring with parafrase. Tech. Rep. 538, Center for Supercomputer Research and Development, Univ. of Illinios, Urbana, 1985.

[16] WOLFE, M. *Optimizing Supercompilers for Supercomputers.* PhD thesis, University of Illinois at Urbana-Champaign, 1982.

[17] WOLFE, M. Advaced loop interchanging. In *Proc. of the 1986 International Conference on Parallel Processing* (1986), IEEE Computer Science Press, pp. 536–543.

[18] ZIMA, H. *Supercompilers for Parallel and Vector Computers.* Frontier Series. ACM Press, 1990.